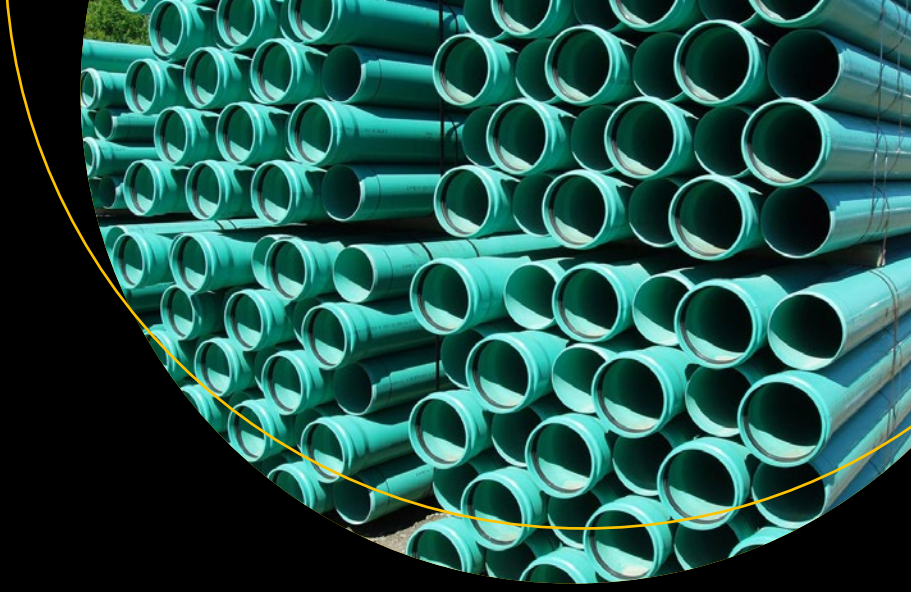


Continuous Integration (CI) and Continuous Delivery (CD)

A Practical Guide to Designing and Developing Pipelines

Henry van Merode

Apress®

Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines

Henry van Merode
Leeuwarden, The Netherlands

ISBN-13 (pbk): 978-1-4842-9227-3 ISBN-13 (electronic): 978-1-4842-9228-0
https://doi.org/10.1007/978-1-4842-9228-0

Copyright © 2023 by Henry van Merode

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Susan McDermott
Development Editor: James Markham
Coordinating Editor: Jessica Vakili
Copy Editor: Kim Wimpsett

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on the GitHub repository: https://github.com/Apress/Continuous-Integration-(CI)-and-Continuous-Delivery-(CD). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author ix

About the Technical Reviewers xi

Acknowledgments xiii

Chapter 1: The Pitfalls of CI/CD 1

Challenges 1

Oversimplified Diagrams and Misalignment 2

Lack of Design Patterns 3

Vulnerabilities 4

Pipeline Testing 4

Application Code vs. Infrastructure Code 5

Organizing and Maintaining Pipelines 6

Technical Constraints 7

Legacy 7

Summary 9

Chapter 2: CI/CD Concepts 11

Principles 12

Positioning of CI/CD 13

Application Lifecycle Management 16

CI/CD Journey 17

Naming Conventions 22

Summary 26

Chapter 3: Requirements Analysis 29

Overview 29

Way of Working 32

Technology 34

Information 39

Security (General) 41

Compliance and Auditability 47

Resource Constraints 58

Manageability 59

Operations 62

Quality Assurance 64

Metrics 68

Monitoring 72

Sustainability 74

Governance 74

Summary 76

Chapter 4: Pipeline Design 77

Design 78

CI/CD and Pipeline Design Approach 79

BPMN 2.0 79

BPMN Elements Overview 80

BPMN in Action 83

Level of Detail 86

Logical Design vs. Realization 87

The Generic CI/CD Pipeline 87

Validate Entry Criteria 89

Execute Build 91

Perform Unit Tests 91
Analyze Code 92
Package Artifact 94
Publish Artifact 94
Provision Test Environment 95
Deploy Artifact to Test 96
Perform Test 97
Validate Infrastructure Compliance 97
Validate Exit Criteria 98
Perform Dual Control 100
Provision Production Environment 101
Deploy Artifact to Production 101
Notify Actors 102
Design Strategies 102
Context Diagram 103
Branching Strategy 105
Build Strategy 125
Test Strategy 139
Release Strategy 158
Production Deployment Strategy 165
Other Design Considerations 183
Summary 205

Chapter 5: Pipeline Development 207

Pipeline Specification 208
Multibranch, Multistage Pipeline 208
User Interface–Based Pipelines 209
Scripted Pipelines 210
Declarative Pipelines 211

Constructs 216
Plugins and Marketplace Solutions 237
Repositories: Everything as Code 237
Third-Party Libraries and Containers 240
Versioning and Tagging 245
Environment Repository 249
Secrets Management 251
Database Credentials 255
Feature Management 257
Development in the Value Streams 260
Simplified Pipeline Development 265
Extended Pipeline Development 266
Advanced Pipeline Development 267
Develop a Base Pipeline 268
Pipeline Generation 270
Pipeline of Pipelines (DevOps Assembly Line) 273
Sustainable Pipeline Development 279
Summary 283

Chapter 6: Testing Pipelines 285

Testing Pipelines 285
Testability of Pipelines 286
Unit Tests 288
Performance Tests 300
Pipeline Compliance and Security Tests 305
Acceptance Tests 307
Summary 307

Chapter 7: Pipeline Implementation 309

Pipeline Implementation 310

Organizational Impact 311

Team Discipline 313

Integration Platform 314

Target Environment Preparations 318

Playbook 318

Application Implementation 319

Runbook 319

Release Note 320

Artifact Promotion 325

Summary 328

Chapter 8: Operate and Monitor 331

Manage the Integration Platform 331

Operational Pipelines 332

Monitor 335

Systems Monitoring 336

Platform Monitoring 342

Business Monitoring 343

Security Monitoring 346

Share Information 349

Events, Alerts, Incidents, and Notifications 352

Summary 356

Chapter 9: Use Case 359

Requirements Analysis 361

Pipeline Design 365

Branching and Release Strategy 367

Release Version Generation 369
Pipeline Development 370
Code Repository 371
Pipeline Creation 375
Configure Variable Groups 376
Configure Service Connections 381
Test 384
Integrity of Artifacts 391
Performance and Acceptance Pipelines 394
Implementation 395
Configure the Azure DevOps Prod Environment and Dual Control 397
Deploy the Application to Production 399
Quality Gate 403
Summary 405

References 407

Index 411

About the Author

Henry van Merode is a solution architect with more than 30 years of experience in ICT within several financial organizations. His experience spans a wide range of technologies and platforms, from IBM mainframes to cloud systems on AWS and Azure. He developed, designed, and architected major financial systems such as Internet banking and order management systems, with a focus on performance, high availability, reliability, maintainability, and security.

For the last 8 years, Henry's expertise has been extended with continuous integration, continuous delivery, and automated pipelines. As an Azure DevOps community lead, Henry likes to talk about this subject and promote automating the software supply chain to the teams at his work.

About the Technical Reviewers

Fred Peek is an IT architect from Utrecht, the Netherlands. He has a master's degree in electrical engineering from the Eindhoven University of Technology. He has more than 20 years of experience in the IT industry, working in software development (Java, C++), software architecture, and security. Besides IT, he is involved in the audio and music industry as a recording/mixing engineer, DJ, and Audio Engineering Society (AES) member.

Joep Daandels is an enthusiastic DevOps engineer from Maaskantje, the Netherlands. He graduated as a software engineer from the Avans University of Applied Sciences and has been working in the IT industry for the last 20 years. In recent years, he specialized in machine learning/artificial intelligence and is currently working on a state-of-the-art solution to enhance IT operations with AIOps. In his spare time, Joep loves to relax and read a good paper or study some interesting new technology.

Ralph van Beek is a DevOps architect specialized in optimizing the CI/CD process for the z/OS mainframe. He graduated in business economics and informatica at Avans Hogeschool for applied sciences. He has been working in the IT industry for 35 years. The last 15 years he has specialized in optimizing and automating software delivery processes for z/OS mainframes, approaching software delivery as a business process. He has been a guest speaker on this topic at various conferences. In his spare time, he prefers various outdoor activities such as hiking and biking and travel photography.

Acknowledgments

Many thanks to the people of Apress for allowing me to write this book and for helping me publish it.

Special thanks to my colleagues Fred, Ralph, and Joep for reviewing the text, providing me with suggestions, and correcting mistakes I made.

And of course, I want to thank my wife Liseth for being supportive.

CHAPTER 1

The Pitfalls of CI/CD

This chapter covers the following:

- The drivers that started my search for a more structured way to design and develop pipelines
- The challenges I faced during the years I worked with continuous integration, continuous delivery, and pipeline development

Challenges

At work, I once gave a presentation about continuous integration/continuous delivery and described how it improves the speed of software delivery. I explained that using pipelines to automate the software delivery process was a real game changer. I presented the theory that was written down in books and articles, until someone from the audience asked me a question about what the development of pipelines looks like and how, for example, one should perform unit tests of pipelines themselves. This question confused me a bit because the theory nicely explains how to unit test an application in the pipeline but never explains how to unit test pipelines themselves. Unfortunately, I could not give a satisfying answer, but this question did make me realize that until then my approach to creating pipelines was a bit naïve and needed a boost. A scan within the department I worked at told me I wasn't the only one who could benefit

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_1

from a more professional approach toward continuous integration/continuous delivery (CI/CD).

In the beginning, I stumbled upon a lot of problems I needed to solve. Unfortunately, there aren't that many tutorials that point out from start to finish how to make a proper pipeline design, which choices to make, which issues I would face, and how to solve them (or at least point me in the right direction). There was no structured way to design, develop, test, and implement pipelines. After a long journey of trial and error, my approach to setting up a CI/CD infrastructure and creating pipelines became more structured and started to show its value. As I watched other teams improve their CI/CD skills, I realized that everybody faced the same challenges and encountered the same pitfalls as I did.

Let me first point out that CI/CD itself is not a pitfall. It is an approach to solving a problem, automating the solution, and implementing a mature software supply chain. But, if underestimated and not understood very well, CI/CD can become a problem that gives you lots of headaches. The realization of pipelines requires a structured approach, similar to designing, developing, testing, and implementing applications, but there isn't much information available that can help you with this journey. My experience is that CI/CD is also not well-understood. Using automated pipelines is not the same as CI/CD. This is one of the reasons why I wrote this book, and this is probably also the reason why you started reading it.

Let me try to emphasize some challenges I faced in the past.

Oversimplified Diagrams and Misalignment

Most CI/CD diagrams depict similar stages like the ones shown in Figure 1-1.

Figure 1-1. *Simplified diagram of CI/CD*

The problem with this type of diagram is that it's fine to explain the concepts of CI/CD, but I noticed that teams use this as their actual blueprint and realize along the way that they have to redesign and rewrite their pipelines. Often, one person is responsible and just starts with a simple implementation of a pipeline without considering the requirements or without even knowing that there are (implicit) requirements. For example, the team works in a certain manner, and that was not taken into account from the start.

The lack of a structured approach to implementing pipelines is one of the underlying problems. The "thinking" processes required before the pipeline implementation starts never happen.

Lack of Design Patterns

Usually, the Internet is a good source for articles, and there are plenty of articles about CI/CD, but questions like "How do I design a pipeline in case the team uses branching strategy X, test strategy Y, and release strategy Z?" remain unanswered. The topics are pointed out to be relevant, but it remains unclear how this translates to the realization of a pipeline.

Most articles are either too abstract or too trivial, or they immediately dive into the technical details, without visualizing the whole picture. There are books about cloud design patterns, enterprise application architecture patterns, and even machine learning patterns, but there is no Gang of Four (see [21]) type of book on CI/CD design patterns. It's a missed opportunity.

Vulnerabilities

Teams are often unaware that they incorporate solutions in their pipeline, which perhaps contain severe vulnerabilities. For example, third-party libraries or software is retrieved directly from the Internet, but from unauthorized sources. This results in a real security risk. Also, the propagation of secrets, tokens, and credentials is often insecure. The CI/CD process should be fully automated, and manually moving secret information around must be prevented. Some of these risks can be avoided, or at least reduced, by applying mitigating actions.

Pipeline Testing

Consider an assembly line of a car-producing company. The company produces cars 24 hours a day, 7 days a week. At the front of the assembly line, the car parts enter. The wheels are mounted to the suspension, the body is placed on the chassis, the engine is installed, and the seats, steering wheel, and electronic equipment are installed. Everything is automated, and at the end of the assembly line, a new car sees the light. What if you are the mechanic who has to replace a large part of this assembly line? Stopping the assembly line is not an option, and replacing assembly line parts while running carries a risk. You may end up with a car with a steering wheel attached to the roof.

This is the underlying problem of the question my colleague once asked when I gave a presentation about continuous integration and continuous delivery, "How do I develop and test my pipelines?" Developing an application and testing it locally works very well for an application, but not so well for pipeline code. The environment in which a pipeline builds the application, deploys it, and executes the tests is not suited to become the develop and test environment of the pipeline itself. And having a local pipeline environment to develop and test the pipeline is often not possible.

Application Code vs. Infrastructure Code

Years ago it was common to set up an infrastructure and install middleware manually. This took ages. You needed various departments to request servers, storage, MQ queues, load balancer configurations, firewall rules, etc. Most of these departments wanted you to fill in an extended request form. If you were lucky, the requested resource was available within five days. Slowly it became more common to automate parts of this process, but today large parts of the on-premises infrastructure still have to be set up manually.[1]

And then I moved to cloud. Working with cloud providers such as AWS and Microsoft Azure opened a new world for me. It was possible to define the whole infrastructure stack using CloudFormation and Azure Resource Manager (ARM) templates. And it became even better. Bicep was introduced for Azure, and AWS introduced the Cloud Development Kit (CDK). This made it possible to program the infrastructure in your favorite programming language. But, this also blurred the dividing line between an application and the infrastructure a bit. Things used to be clear. Scripts, which were used to create (parts of) the infrastructure or middleware, were not validated against quality rules and were not tested in the pipeline. The pipeline was focused on the application. With the introduction of infrastructure as code (IaC), the pipeline should treat infrastructure code similarly to application code. Infrastructure code must be validated against security policies and organization guidelines. The provisioned infrastructure must be "tested" to make sure that the target environment behaves the way it was intended. This indicates that IaC has an important role in a pipeline.

[1] Organizations with on-premises datacenters are trying to catch up slowly, implementing platforms such as OpenShift, disclosing their infrastructure using APIs, and making use of Ansible and Terraform to define infrastructure as code.

Note I used to see infrastructure provisioning and application deployment as two different processes. However, this is "legacy thinking" and is based upon the fact that this was often done by different departments and different people, using different technologies. But if you think about what you want to achieve, it makes sense not to see this as different processes. A deployment is putting a business capability into production. It does not matter if this involves infrastructure, an application, security policies, or monitoring for that matter. All these components together contribute to the business capability, and they should not be treated as different "things." This becomes more apparent if all resources become "virtual" and are defined as code.

Organizing and Maintaining Pipelines

After the first pipeline is created, the second one is created and then the third and the fourth, until there are dozens of pipelines trying to realize one or more steps within a CI/CD process. Often code is copied from one pipeline to the other, making it more complex to maintain. After a while, a complete restructuring is needed. There is no vision from the start. Various factors influence the organization of pipelines.

- The application architecture is important. Is the application a monolith, or does it consist of multiple microservices? In how many pipelines does this result?
- What is the teams' workflow? Do they use a particular branching strategy, and what is their test strategy, their deployment strategy, and their release strategy?

- Also, the arrangement of the application code, the infrastructure code, and the pipeline code requires some thinking. Do you put everything in one repository or not?
- Does the pipeline include manual steps, and how do they reflect in the design?

The number of pipelines may grow if the number of variations is high. I've seen examples in which one small application resulted in multiple pipelines: one pipeline performing just the CI stage for feature branches, one pipeline for a regular (snapshot) build, one for a release build, one deployment pipeline to set up tests, a separate pipeline to perform the—automated—tests, and a deployment pipeline for the production environment.

Technical Constraints

Something that comes to the surface only after a while is that you may hit some constraint, often a compute or storage resource. For example, the code base becomes bigger and bigger, and the source code analysis stage runs for hours, basically nullifying the CI/CD concept of fast feedback. Also, the queuing of build jobs may become an issue in case the build server cannot handle that many builds at the same time. Unfortunately, these constraints are often difficult to predict, although some aspects can already be taken into account from the start.

Legacy

If we like it or not, a lot of teams still use a legacy way of working. They still perform too many manual tests, and test environments are often set up manually. As a branching workflow, Gitflow is still used a lot. This type of workflow has a few downsides. It is complex, with multiple—long-lived—branches, and it can be slow to adapt new features because of a strict release cycle.

But even if an organization starts to innovate and implements the principles of CI/CD, there is still a long way to go. People have to realize—and sometimes be convinced—that things can be improved. They need to understand how this can be done, and they need to know their role in this process. CI/CD is not just about the tools, although there were many times I put a curse on some of them. With most of the current tools, you can quickly build a simple pipeline in a couple of hours. The challenge is about shaping the organization, the processes, and the people. Do not underestimate how much effort that takes, especially in a large organization.

So, should you send the teams that match the previous description the message that they aren't ready yet for CI/CD and leave it to that? Of course not. I always see CI/CD as a "growing model." Start small, extend gradually, inspire people, and help shape a way of working that fits in the CI/CD philosophy.

These challenges started my search for a structured CI/CD design approach, but without much success. My CI/CD journey, therefore, consisted of a lot of trial and error, but I did manage to learn a couple of things along the way. Coming mainly from a Java/WebLogic/WebSphere/Spring Boot environment, I designed and built Java-based pipelines for multiple teams and mastered a bunch of tools that help with automating builds, setting up test and production environments, deploying applications, and executing automated tests. Tools like Jenkins, Azure DevOps, and Ansible come to mind.

I learned what worked and what did not. I realized that more traditional workflows do not fit very well in the CI/CD concept. I experienced that the development of pipelines does not always seem like a team responsibility; I often heard team members talking about "your pipelines." So, turning this situation around and making CI/CD a shared responsibility was also part of the job. Slowly, the approach to designing, developing, and implementing CI/CD pipelines became more structured. It would be a shame not to share my experiences.

Note This book is a practical guide to designing and developing pipelines. The ambition is geared toward CI/CD, but the scope is a bit broader. As explained, no team can switch to "pure" CI/CD in an instant, so to accommodate these teams, the book also discusses workflows that include traditional branching strategies, for example.

Summary

You learned about the following topics in this chapter:

- Teams must be aware of the challenges they face when they start with continuous integration and continuous delivery.
- These challenges involve the following:
  - The use of oversimplified diagrams as a blueprint for pipelines
  - No clear design patterns
  - Vulnerabilities in pipelines
  - Testing pipelines
  - The use of infrastructure as code in pipelines
  - Managing pipelines
  - Technical constraints
  - Legacy and pipelines

CHAPTER 2

CI/CD Concepts

This chapter covers the following:

- The principles of continuous integration and continuous delivery
- Continuous integration and continuous delivery in the context of the Open Groups' IT4IT Reference Architecture and the software supply chain
- The importance of application life-cycle management (ALM), with examples of ALM platforms
- The bumpy journey of realizing continuous integration and continuous delivery
- Pipeline development with application development (this forms the basis of this book)
- The importance of a thorough requirements analysis, which forms the basis of your pipeline design and development
- Terms often used in continuous integration and continuous delivery and what they mean

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_2

Principles

The foundations of continuous integration/continuous delivery (CI/CD) were laid down by people like Paul Duvall, Jez Humble, and David Farley, and they are thoroughly described in their respective books, *Continuous Integration: Improving Software Quality and Reducing Risk* (see [5]) and *Continuous Delivery: Reliable Software Release through Build, Test and Deployment Automation* (see [6]). These books present a couple of concepts and principles that together make up CI/CD. Let's try to summarize CI/CD in a few sentences.

The benefit of continuous integration and continuous delivery is that application code can be delivered faster to production by automating the software supply chain. This produces secure code of better quality, provides faster feedback, and results in a faster time to market of the product.

Continuous integration is based on the fact that application code is stored in a source control management system (SCM). Every change in this code triggers an automated build process that produces a build artifact, which is stored in a central, accessible repository. The build process is reproducible, so every time the build is executed from the same code, the same result is expected. The build processes run on a specific machine, the integration or build server. The integration server is sized in such a way that the build execution is fast.

Continuous delivery is based on the fact that there is always a stable mainline of the code, and deployment to production can take place anytime from that mainline. The mainline is kept production-ready, facilitated by the automation of deployments and tests. An artifact is built only once and is retrieved from a central repository. Deployments to test and production environments are performed in the same way, and the same artifact is used for all target environments. Each build is automatically tested on a test machine that resembles the actual production environment. If it runs on a test machine, it should also run on the production machine. Various tests are performed to guarantee

that the application meets both the functional and the nonfunctional requirements. The DevOps team is given full insight into the progress of the continuous delivery process using fast feedback from the integration server (via short feedback loops).

This is a concise explanation of CI/CD, which is of course more thoroughly described in the mentioned books.

Positioning of CI/CD

The IT value chain provides a view of all activities in an organization that create value for the organization [7]. The IT value chain concept is defined in the Open Groups' IT4IT Reference Architecture and consists of four pillars, the value streams.

- *Strategy to portfolio (S2P) value stream*: Aligns the IT and business road maps and includes activities such as setting up standards and policies, defining the enterprise architecture, analyzing service demand, and creating service road maps.

  CI/CD practices are implicitly used to support the portfolio management process by providing a consistent and repeatable way to build, test, and deliver new IT investments. Additionally, CI/CD can help to ensure that IT investments meet required standards for quality, security, and compliance by automating the testing and deployment process and by providing visibility into the status of the delivery pipeline.

- *Requirement to deploy (R2D) value stream*: Provides the framework for creating/sourcing new services or modifying those that already exist. This value stream includes the typical activities to create the services, planning, requirements analysis, design, development, test, and deployment.

In essence, this is what CI/CD is mainly about. CI/CD pipelines implement the activities associated with the R2D value stream.

- *Request to fulfill (R2F) value stream*: Provides a framework connecting the various consumers (business users, IT practitioners, or end customers) with goods and services that are used to satisfy productivity and innovation needs. This typically includes activities of supporting departments that deliver facilities, tools, and automation support, which help DevOps teams in the development of their services.

  Managing a CI/CD SaaS solution, developing a CI/CD infrastructure, and hosting a CI/CD platform infrastructure are typical examples of CI/CD activities in the R2F value stream.

- *Detect to correct (D2C) value stream*: Deals with integrating monitoring, management, remediation, and other operational aspects. Key activities are detecting events, alarming, diagnostics to determine root causes, determining business impact in the case of issues, and resolving incidents.

CI/CD helps to ensure that the fix or update is delivered to the customer quickly and with a high level of quality.

The IT value chain is more or less a set of interrelated activities that organizations use to create a competitive advantage. It is valuable in the sense that it consists of a thorough list of activities that can be mapped on the software supply chain.

The software supply chain represents activities required to get the product to the customer. It is a subset of the IT value chain activities, but more targeted toward the process of idea creation until the actual

service rollout. CI/CD covers activities of the software supply chain with a focus to speed up software development and to maintain a high-quality standard.

Traditionally, CI/CD does not cover all activities associated with software development. CI/CD is usually restricted to build, test, and deployment. Activities such as planning, requirements analysis, designing, operating, and monitoring are usually not considered in the scope of CI/CD; however, we shouldn't be too narrow-minded here. It does make sense to keep these activities in mind when realizing pipelines.

Consider the case in which an artifact is deployed to production. It needs to be monitored. Incidents may occur, which need to be resolved. What if application monitoring becomes integrated into the pipeline? Issues and incidents detected by the monitoring system could lead to the automatic creation of work items, or it could even lead to automated remediation; an incident detected results in triggering a pipeline that remediates the incident. Stretching this thought process a bit more and anomalies detected by artificial intelligence (AI) monitoring may result in triggering a pipeline that reconfigures a service even before the incident occurs.

It is good to see in practice that some teams stretch their CI/CD pipeline setup to the max, looking beyond the scope of traditional CI/CD and considering all steps in software development.

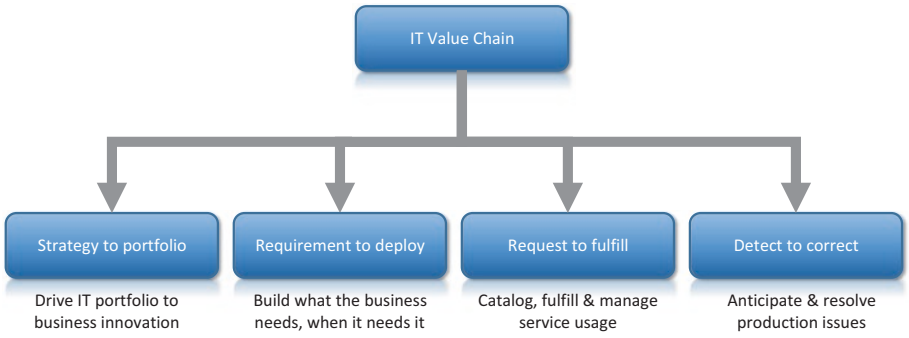

IT Value Chain
Strategy to portfolio
Drive IT portfolio to business innovation
Requirement to deploy
Build what the business needs, when it needs it
Request to fulfill
Catalog, fulfill & manage service usage
Detect to correct
Anticipate & resolve production issues

Figure 2-1. *IT value chain*

Application Lifecycle Management

One of the earlier continuous integration tools was Hudson, maintained by Oracle and later forked to what is currently known as Jenkins. With its success, new tools arrived like Travis CI and Circle CI, each extending the CI concepts. In addition, specific deployment tools popped up, covering the continuous delivery part of the CI/CD equation. A tool like Octopus Deploy is such an example. To cover even more aspects of the software supply chain, the toolset expanded with issue trackers and monitoring tools.

The problem was—and still is—that integration of all these tools is not straightforward, and CI/CD requirements cannot always be implemented easily. This means that time (and money) must be spent to create a fully integrated toolset, which not only implements all functional requirements but is also performant, secure, and stable.

Here is where application life-cycle management suites step in. ALM tools include portfolio management, project management, requirements management, software architecture, application development, continuous integration, quality assurance/software testing, software maintenance/support, change management, release management, and monitoring. It covers more than only the software development life cycle and focuses on the whole software supply chain. Examples of ALM suites are Azure DevOps, a software-as-a-service (SaaS) solution from Microsoft, and the Atlassian ALM suite consisting of Confluence, Bitbucket, Sourcetree, HipChat, Jira, and Bamboo.

And even these current-gen ALM platforms cover only parts of the software supply chain, or they omit certain functionality, which means that these features must still be added using additional tools, marketplace solutions, or DIY solutions.

Note An ALM platform is a collection of tools and processes that support the various stages of an application's life cycle. Specifically, the software development life cycle (SDLC) tools of the platform play a crucial role in the context of CI/CD processes.

Throughout this book, the name *ALM platform* is used. This can also be read as an integration server, a build server, or a set of individual but integrated CI and CD tools.

CI/CD Journey

I do not know one team that implemented CI/CD in the first iteration. When I ask a team to think about a solution to deliver software in smaller increments and more frequently, they agree it is a good idea but difficult to realize in their context. They give various reasons why this is not possible or at least very difficult. A generic problem seems to be that teams are used to a certain way of working, often a way of working that does not necessarily meet the preconditions of a CI/CD implementation. They find it hard to let go, especially if the new way of working is not crystal clear to them or if they don't realize the necessity to change. And even if they realize it, they still need to adapt. Change remains difficult.

A recurring problem, for example, deals with the granularity of user stories or tasks. Some stories or tasks are just put down as one-liners, like "implement the validation of a digital signature." A developer commits to this story and starts coding.

This is what happens: After the validation code is written, it needs to be tested. This requires additional test code to be written. The test code is needed to create the digital signature that needs to be validated. But testing also requires a key pair and a certificate. The key pair and a certificate signing request (CSR) file are created, and the certificate is obtained from

the local public key infrastructure (PKI) shop (assuming that self-signed certificates are not allowed in this company). The developer also realizes that the target environment does not have a file system but an object store. Storing the certificate on the workstation's file system works fine for local testing, but it does not work anymore after the code has been integrated into the app and deployed to the target environment. So, the code has to be rewritten, and by the way, additional measures have to be taken from an access control point of view, so the app can also read the object store. The story looked simple at glance but expands along the way. The result is that the developer keeps the code and pushes it to the central repository only after a couple of days, or even longer. The translation from business requirements to epics, stories, and tasks is not trivial, and decomposing the work into small, manageable chunks is often a challenge.

Realizing that implementing CI/CD is a journey is the first step of the transformation process. It is the first hurdle of a bumpy journey. Setting an ambition level helps in defining this journey. Team members should ask themselves a couple of questions. Where do we stand six months or one year from here? What can be improved in our way of working? What do we need to fix certain impediments our team deals with? Can they be solved by training?

Determining the ambition level can be done with the help of a continuous delivery maturity model. This model helps assess the team's current maturity and works as guidance in their CI/CD journey. There are several examples of continuous delivery maturity models. The following one is from the National Institute for the Software Industry (NISI; see Reference [36] and Figure 2-2). The vertical axis represents the categories or steps in software development. The horizontal axis represents five maturity levels, from *foundation* to *expert*. These maturity levels indicate how well a team performs in its continuous delivery practice. It is up to the team—also driven by the organization's ambition—to decide in which areas they need improvement and to what extent. Maybe they don't want to be an expert in each category. Create an initial road map, but start small and expand over time.

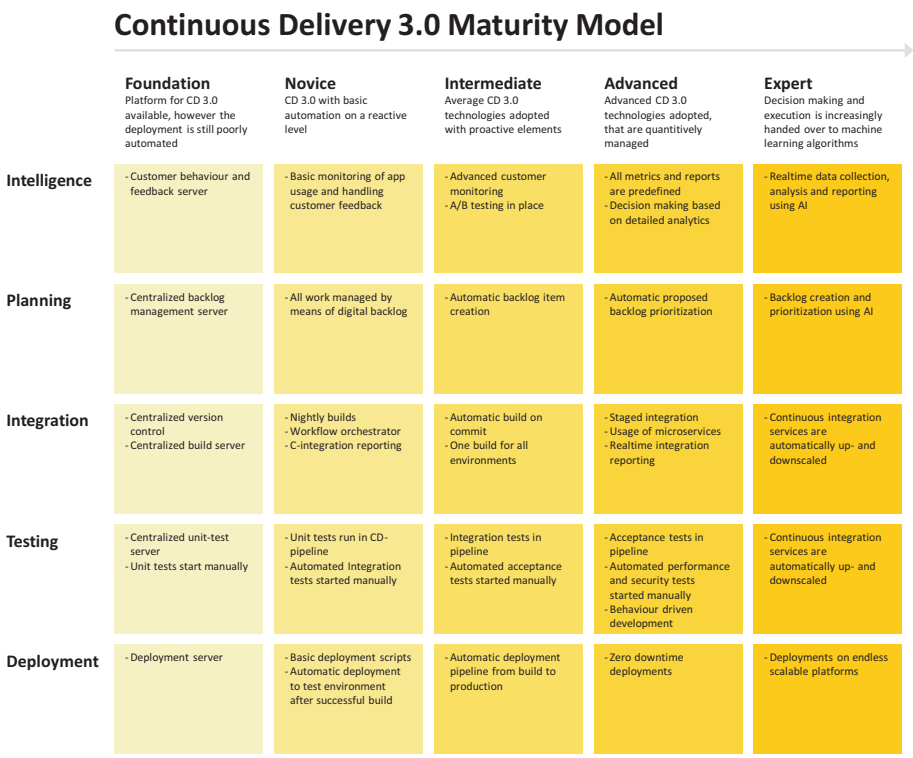

Continuous Delivery 3.0 Maturity Model
Foundation
Platform for CD 3.0 available, however the deployment is still poorly automated
Novice
CD 3.0 with basic automation on a reactive level
Intermediate
Average CD 3.0 technologies adopted with proactive elements
Advanced
Advanced CD 3.0 technologies adopted, that are quantitively managed
Expert
Decision making and execution is increasingly handed over to machine learning algorithms
Intelligence
- Customer behaviour and feedback server
- Basic monitoring of app usage and handling customer feedback
- Advanced customer monitoring
- A/B testing in place
- All metrics and reports are predefined
- Decision making based on detailed analytics
- Realtime data collection, analysis and reporting using AI
Planning
- Centralized backlog management server
- All work managed by means of digital backlog
- Automatic backlog item creation
- Automatic proposed backlog prioritization
- Backlog creation and prioritization using AI
Integration
- Centralized version control
- Centralized build server
- Nightly builds
- Workflow orchestrator
- C-integration reporting
- Automatic build on commit
- One build for all environments
- Staged integration
- Usage of microservices
- Realtime integration reporting
- Continuous integration services are automatically up- and downscaled
Testing
- Centralized unit-test server
- Unit tests start manually
- Unit tests run in CD-pipeline
- Automated Integration tests started manually
- Integration tests in pipeline
- Automated acceptance tests started manually
- Acceptance tests in pipeline
- Automated performance and security tests started manually
- Behaviour driven development
- Continuous integration services are automatically up- and downscaled
Deployment
- Deployment server
- Basic deployment scripts
- Automatic deployment to test environment after successful build
- Automatic deployment pipeline from build to production
- Zero downtime deployments
- Deployments on endless scalable platforms

Figure 2-2. *NISI continuous delivery maturity model*

Important in the CI/CD journey is that it must be a team effort. There are enough cases in which one or two team members are assigned to "implement CI/CD." The danger exists that they become too isolated, and any question about the topic is immediately delegated to them. If the team is not involved, the knowledge gap becomes bigger and bigger, and when they leave the team, there is nobody left to take over. So, involve the whole team and relevant stakeholders and take them along on the CI/CD journey. And, of course, not everybody needs to know every nitty-gritty detail, but it must be enough to understand what's going on. A good practice is to keep CI/CD realization in sync with the workflow of the team.

Add CI/CD-related work items to the sprint and keep the same pace as the rest of the team. Give a sprint demo from time to time. Involve other team members to take up small bits and pieces, once the pipelines are mature and more or less stable.

Until now, CI/CD is presented as an abstract concept with a certain philosophy, but a concept does not run on a real server. The implementation of CI/CD also involves running pipelines that build, test, and deploy software. The pipelines themselves are pieces of software running on a server. This statement forms the basis of this book; a pipeline is software. So, why shouldn't you treat pipeline development the same way as developing an application? With this in mind, consider the steps of software development.

- *Requirements analysis*: The first step in software development is the requirements analysis phase. In our context, it involves gathering requirements to understand the problem domain of CI/CD. This also helps in scoping the implementation.
- *Design*: Designing pipelines is the process that helps you understand the flow of the pipelines. It makes clear which conditions to consider and where the pipeline takes an alternative path. A design also helps to determine which tasks are executed and where they fit best in the pipeline. The design also visualizes which external systems are involved and how the pipeline communicates with them.
- *Development*: This concerns the actual development of pipelines and the integration with other tools and surrounding systems.
- *Test*: The context here is about testing the pipelines themselves, not testing an application within a pipeline. Pipelines are also software, and the

preconditions to test a pipeline differ from the preconditions required to test an application. Is there a test environment specifically for pipeline testing? What kind of tests are involved, and is it perhaps possible to automate these tests?

- *Implement*: Before the pipelines can be used, all interfaces with external systems are set up, the configuration is performed, variables are set, and monitoring dashboards are created. The team is prepared, and knowledge is shared. Any remaining issues and improvements have been discussed.
- *Operate and monitor*: The behavior of the pipelines is checked in the operating and monitoring phases. Does queuing happen, and is the overall execution time of the pipeline in order?

This book describes how pipelines are designed and developed from the viewpoint of software development. Each chapter covers one phase of the pipeline development process, but on an abstract or semitechnical level. It provides a structured approach to design and develop pipelines. The final chapter dives into a use case and uses the strategies of all previous chapters to design and develop pipelines using Azure DevOps in combination with AWS. The code used in this chapter is provided as research material.

If you are looking for an in-depth technical—how-to—book about the development and implementation of pipelines using specific tools like Jenkins or Azure DevOps, this is probably not the book you are looking for. However, if you are looking for guidelines on how to start with CI/CD, how to design the process and the associated pipelines, and what needs to be considered during the development and implementation of pipelines, this book is for you.

Naming Conventions

There is no "standard" glossary for CI/CD, and sometimes the same name is used in a different context. For example, *deploy* is also referred to as *release* (the verb), but *release* can also refer to the creation of a *release (candidate)*, as in the noun.

So, to avoid confusion, this book provides the following definitions. Note that this is not an exhaustive list. Only the words that need explanation or that might cause confusion are listed.

Analyze code: This is a subset of quality assurance and includes static code scans to determine code quality and detect vulnerabilities in the application code and its dependencies.

Application life-cycle management (ALM): This integrated toolset covers the main aspects of the software supply chain.

ALM platform: This is either a real ALM platform, an integration server, or a set of individual but integrated CI and CD tools. The ALM platform covers the complete CI/CD toolchain.

Artifact: An artifact is a package stored in a binary repository and used for deployment to a target environment.

Branch: This is a branch used in source control management.

Build: This means combining source code and its dependencies and creating a runnable product (artifact).

Continuous deployment: This is a process in which the artifact is built, tested, and deployed to production unattended. The pipeline validates whether the artifact meets all quality criteria. This is in contrast to continuous delivery where an extra manual step (dual control) is needed.

Deploy: This means installing an artifact on a certain target environment. This can be a test environment or a production environment.

Dual control: This is the application of the four-eyes principle in which one person performs a task and another person has to approve the execution of that task.

Environment: In most cases, this refers to the platform/infrastructure on which the artifact is deployed. In some cases, the environment is used in the context of an ALM platform and/or related CI/CD tools.

Package (verb): After an application is built, it is packaged in a way to make it easily transportable, like a `.zip` file or a `.jar` file. In the case of commercial off-the-shelf (COTS) products, from a consumer point of view, this stage is usually skipped because it is already in a transportable format. Packaging also implies baselining the artifact, to make sure that what is deployed to production is indeed the proper artifact (tested and uncompromised).

Package (noun): This is the artifact, built by the integration server or a downloaded file from a vendor in the case of a COTS application.

Pipeline: This is the design and implementation of all steps that define the automation of the software delivery process. This can be either a real CI/CD pipeline or a pipeline with a less continuous character.

Promote: This activity "promotes" a release candidate to a release. The release can be deployed to a production environment. Sometimes this is an implicit step: "we now call it a release." But in some ALM platforms it is an explicit task.

Publish: After the application has been packaged, it is stored in an immutable binary repository, such as Artifactory or Nexus. Downloaded packages from vendors (COTS) still need to be published to a secure location within the organization to guarantee integrity and traceability.

Quality assurance: This process makes sure the quality of your product meets a certain level.

Release (verb): This is when activities are performed to deploy an artifact to a production environment.

Release or release version (noun): This is an artifact that can be deployed to a production environment.

Release build: This is the creation of a release candidate.

Release candidate: This is an artifact to which no new features are added anymore. Only bug fixes are solved in a release candidate, so it becomes a new release candidate again, but with a different version. If all bugs are fixed and the release candidate passes all tests, it is promoted to a release (the noun).

Snapshot build: This is an old remnant of the Maven workflow. A snapshot build will never find its way to production. It is an intermediate build artifact from a feature or the develop branch and used as input for a pull request (if a snapshot build fails, the code is not merged).

Software development life cycle (SDLC): This is the process of software development, build, test, and deployment.

Source control management (SCM): This is a system to perform version control. Examples of SCM systems are Git, Mercurial, and Subversion.

Stage: This is a group of related activities in a CI/CD pipeline. Examples of stages are *Execute build*, *Analyze code*, and *Perform test*.

Tag: A tag is used to identify a group. This group can consist of code, artifacts, stages, target resources, etc. A tag is often used to identify a release (candidate), with all its related code, CI/CD stage(s), and target resources.

Task: This is one activity in a stage. A testing stage for example can consist of different test types, like a regression test or a preproduction test, each performed as a task.

Target/target environment: This is the environment in which an artifact is deployed. The target and target environment are not used in the context of an ALM platform and/or related CI/CD tools.

The target environment is either a test server or a production server. It can also be an account/subscription on a cloud service provider.

Test: Testing is a subset of quality assurance and involves automated testing and manual testing.

Trunk: This is the main branch in a source control management system. It is also called mainline or master.

Versioning: Artifacts need to be versioned to identify them. A version is something different than a tag. The version refers to a specific instance of the artifact, and a tag is applied to a group of which the artifact is one. The version and tag can have the same value, but this is not mandatory. A tag may refer to a product feature that is associated with several release candidates, each having its version.

Summary

You learned about these topics in this chapter:

- A brief overview of continuous integration and continuous delivery outline
- Positioning of continuous integration and continuous delivery in the software supply chain
- Application life-cycle management (ALM)
- The journey of implementing continuous integration and continuous delivery

- The process of pipeline development is the same as the development of an application and involves the following:
    - Requirements analysis
    - Pipeline design
    - Pipeline development
    - Testing pipelines
    - Pipeline implementation
    - Operate and monitor pipelines
- Keywords associated with continuous integration and continuous delivery

CHAPTER 3

Requirements Analysis

This chapter covers the following:

- The sources of pipeline requirements
- Various categories of requirements
- Requirements in detail

This chapter is intended to inspire you by presenting an overview of requirements, grouped by category.

Overview

Requirements analysis is the first step before the actual design of the pipeline is drafted and the pipeline is created. Requirements apply to CI/CD practices, pipelines, the ALM platform, or a combination of all tools that make up the integration infrastructure. Requirements are derived from different sources.

- First, there are basic CI/CD principles, which can be treated as requirements. Become familiar with them. If you deviate from the basic principles, you must have a good reason to do so because they form the foundation

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_3

of CI/CD. Sources like [5] and [6] are excellent in explaining these principles and helping you grasp the concepts of CI/CD. To emphasize, CI/CD itself is not the goal. Delivering quality software at a pace that satisfies your business is the goal. This could mean that you deviate from the continuous delivery "theory" in certain aspects.

- There are also best practices. The Internet is full of them. Some are useful, others are not, but sometimes they can be helpful sources. Understand these best practices. A nice source of best (good) practices is, for example, described in [8].
- The business organization has requirements to which a pipeline must comply. Often this relates to the way of working in an organization or to specific security constraints. This poses requirements for the design and implementation of a pipeline. Organization requirements are usually published somewhere on an intranet site. Make sure they are known and understood.
- The DevOps team has requirements. Some of these requirements are explicit, such as "we want a dashboard of all artifacts and versions deployed on all test environments." Some of the requirements are more implicit. For example, the team might adopt a certain branching strategy, like a trunk-based workflow, which poses requirements to the pipeline. The team maybe has a certain way of testing. Test engineers perform manual tests on their local workstations in addition to automated tests in a dedicated test environment. How does this translate to a pipeline? Gathering the team's requirements is essential for a good pipeline design.

Requirements analysis covers various areas, as listed in Table 3-1.

Table 3-1. *Requirements Analysis Areas*

Way of working	Resource constraints	Monitoring
Technology	Manageability	Sustainability
Information	Operations	Governance
Security (general)	Quality Assurance	
Compliance and Auditability	Metrics	

This list is not exhaustive but gives an idea of which areas must be considered. Of course, more areas can be identified, and some maturity models define areas such as business intelligence, planning, culture, and organization. These maturity models list some expert/advanced capabilities such as *automated remediation based on (AI) monitoring* and *automated prioritization of the backlog based on AI*. However, this book intends to give practical guidelines and not an advanced vision of CI/CD because most companies will never reach that level. Moreover, in practice, it is not even always possible to achieve a complete hands-off software supply chain with all the bells and whistles. Just think of manual intervention by operators because certain situations are not foreseen and cannot be solved using a pipeline. Also, costs play an important role in the realization of an automated software supply chain. This means you always have to make a weighted choice between requirements that are absolutely necessary and requirements that are not.

The remaining pages of this chapter describe the areas mentioned in Table 3-1 in more detail and show some examples of requirements that are worth checking out.[1] These requirements serve the purpose to inspire and

[1] I tried to prevent being Captain Obvious. A lot of requirements are implicit and part of CI/CD practice, such as "tests are automated" and "use version control," so they are not listed explicitly.

increase awareness of what is possible, what is important, what works for you, and which requirements are not worth considering right now.

The following are possible requirement suggestions—grouped per area—with each requirement to be validated for relevance and applicability to your specific situation. Note that the order of topics in this chapter has some degree of randomness and says nothing about how important a requirement is. Additionally, some requirements may feel like they immediately delve into the subject matter without any introduction. Do not worry. In the following chapters, we will take a closer look at the topics.

Before we go into more detail, it is important to stress one (meta) requirement that applies to all requirements:

Requirement: A requirement has an owner.

It has happened too often—to me and my colleagues—that someone emphatically introduced a requirement, which entailed a lot of inflexibility and costs, but where no one could concretely explain why this was necessary. A requirement without an owner and justification is not a requirement. Implement a requirement only if there is a need to do so.

Way of Working

The way of working can be defined on a business organization level or team level. It defines the following:

- *The way of working of the business organization*: The business organization may use Agile and Scrum, biweekly sprints, or multiple DevOps teams working on the development of one product. In some way, these aspects influence the pipeline design.

- *The team's branching strategy*: The team's branching strategy plays an important, and even bigger, role. The CI/CD process, and therefore the pipeline design, strongly depends on the workflow of the team; do they use a trunk-based workflow, a feature branch workflow, or the "old-fashioned" Gitflow?
- *The test strategy of the team*: A CI/CD process consists of numerous types of tests, some continuous, others less continuous. Examples are unit tests, integration tests, functional tests, regression tests, manual tests, load tests, stress tests, performance tests, break tests, and preproduction/staging tests.
- *Release strategy*: This defines the cadence to release artifacts and deploy them to production.
- *The production deployment strategy*: In addition, the production deployment strategy shapes the pipeline design. Does the team use a "Re-create deployment" strategy or a "Blue/Green deployment" strategy?

Requirement: Use a simple branching strategy.

The more complex a workflow is, the less "continuous" the workflow is, and the more complex pipelines become. Limit your branching strategy to a trunk-based workflow or feature branch workflow, which is described in successive chapters.

Requirement: Keep feature branches short-lived.

One of the basic principles of CI/CD is not to use feature branches, but if you decide to use them, keep the feature branches short-lived. The longer a feature branch is under development, the more difficult it becomes to merge other features back to the trunk (or another branch) because that branch was significantly changed.

Requirement: Choose the release strategy you want, but keep the mainline production-ready.

Deploying a release once a day, once a week, or once a month is a requirement the business defines. They probably have good reasons to release either very often or with larger time intervals. This does not matter. But it is good practice always to keep your mainline in such a state that it is possible to deploy whenever you want. Even if you release once a month, you are still practicing the CI/CD principles if the mainline is in a production-ready state.

Requirement: Perform manual testing only if needed.

Performing manual testing is a CI/CD anti-pattern, but practice shows that manual testing or semi-automated testing is still required. The following are the reasons why:

- The QA team has a backlog converting manual tests to automated tests.
- The automated test of a newly developed feature is not yet integrated into the automated test suite. The trick is therefore to integrate manual testing somehow into the CI/CD process.
- Automating the test is costly if this particular test is rarely executed.
- Some tests are very specific, so they cannot be automated. Usability testing is such an example.

Technology

The target environment, the CI/CD framework, the tools, and the application architecture all influence the realization of a pipeline and its flow. Here are a few examples:

- A microservice runs independently. This means the build and deployment process must also be independent of other microservices.
- Often, on-premises target environments are manually created, or the creation is only partly automated. Cloud service providers like AWS or Azure offer a lot of possibilities to create ephemeral test environments, which are test environments that are automatically created and destroyed on demand. Tasks to create test environments should be embedded in the pipeline.
- There is a huge difference in the process of self-built applications compared to commercial off-the-self (COTS) packages. CI/CD for vendor packages even sounds like a contradiction from a consumer point of view, and in essence, it is. But that does not prevent anyone from creating an automated pipeline that supports the download, validation, deployment, and configuration of COTS packages.

Requirement: The availability, integrity, and confidentiality of the ALM platform/integration server must match those of the application with the highest classification.

If the deployed application is highly available, has the highest integrity classification, and processes data that must be treated as confidential, what does this mean for the software supply chain?

Take *availability*, for example. If an incident occurs and the application must be fixed immediately, the ALM platform or integration server should be available. But if this is not the case, the fix cannot be deployed. There are a couple of options.

- Accept the risk. What is the chance that the ALM platform/integration server is not available at the same moment an incident with the application occurs?
- Always have a *desired path*. This is an alternative shortcut to bypass the pipeline. The use of this shortcut must be regulated with strict security measures, of course.
- The other alternative is to increase the availability of the ALM platform/integration server so that the RTO of the ALM platform/integration server matches the RTO of the application. From a risk and security point of view, this is the best solution. From a cost point of view, probably not.

Similar requirements apply to the integrity and confidentiality of an application. How secure is an application if the tools and libraries used to create and deploy the application are not secure (enough)? The requirements analysis sections "Security (General)" and "Compliance and Auditability" take a closer look at this topic.

Note An ALM/integration platform runs pipelines of multiple applications. Even if just one application has the highest availability/integrity/confidentiality classification, the platform should comply with this classification.

Requirement: Create a pipeline per microservice.

A microservice is a small, isolated piece of software that runs independently. The goal is that teams can bring them into production independent of other microservices. This implies that a microservice must have its CI (build) and CD (deployment) pipeline. One solution is to use a base pipeline template or libraries to generalize the pipeline and extend from the base pipeline for each microservice.

Requirement: Automate the creation of ephemeral test environments.

Manually created environments take too much time and introduce the risk of differences between test and production environments. Keeping permanent environments—when not used anymore or not used often—are costly. In addition, automated tests should be able to run independently, which justifies a dedicated test environment.

The use of ephemeral test environments is a solution to this problem and is strongly recommended. The test environment is created on the fly and automatically destroyed again when not being used anymore.

Requirement: Don't re-create test environments in every pipeline run.

Although cloud service providers provide all the tools that enable the creation and deletion of test environments on the fly, it is not a very good approach to do this every time the pipeline runs or after every SCM push. Creation and deletion of a test environment cost time—even in the case of a cloud provider, this can take half an hour—which adds up to the overall execution time of the pipeline. In addition, creating and deleting a test environment every time the pipeline runs becomes costly in terms of money.

A better approach is to provision the infrastructure but delete it only if not used anymore or if not been used for a longer time. Fortunately, the facilities to create an environment—in the cloud—are idempotent, so running the provisioning of infrastructure resources multiple times does not change the test environment if the infrastructure code has not been changed. And instead of deleting the whole test environment, it may be useful to include tasks in the pipeline that reset the test environment to a certain status, after a test was executed (e.g., reset the data in a database table).

Requirement: Develop an automated pipeline for COTS applications.

Even in the case of vendor packages (commercial off-the-shelf), the use of an automated pipeline has various benefits. The execution of downloading the software, validating, installing in the test environment, configuring, verifying, and installing the software in production is orchestrated and repeatable. In addition, ALM platforms often have built-in features that make the audit trail visible.

Requirement: Use the same OS in the pipeline as the runtime environment OS.

An application may behave differently if built on another OS than the OS for which it was developed. Make sure that it is built and tested on the same OS also used in the production environment.

Requirement: Use an ALM platform and limit the number of additional tools.

In general, a large range of tools is difficult to integrate, and most modern ALM platforms already consist of features integrated within one platform. Use additional tools only if the platform itself does not support a certain feature or if the capabilities of the platform's feature are too limited. Using a limited set of tools has some other advantages.

- Keeping the number of tools limited means less maintenance.
- Knowledge is more consolidated.
- Additional tools must be assessed to prove they comply with the organization's security policies.
- Fewer tools also mean fewer licenses and potentially fewer costs.

The choice of what platform to use depends on the type of organization. In some cases, this choice is made at the organizational level, and teams have to adhere to this choice. In other situations, the team itself decides; if they feel comfortable with GitHub actions, for example, they are more likely to choose this option.

Information

ALM platforms potentially generate a lot of data that the team can use to keep informed. Even if the platform has default overviews and notification options, it still makes sense to think about how a team is informed and what type of information is shared with the team. Often these tools send a lot of emails, which results in team members not reading their emails anymore. Information overloading is a common problem and must be managed using several strategies.

- *Information pull and push*: What type of information is important enough to push to the team members in the form of a notification—such as an email—and what type of information is not? In the latter case, a team member can also actively search for information if it is needed.
- *Display capabilities*: Overviews in some ALM tools don't always excel in readability. The overview is often cluttered with all types of build and deployment information. Sending the information to alternative tools that provide different views and/or have better displaying and filtering capabilities may be something to consider.
- *Channel*: Preferably use a limited number of options to inform teams. One tool to push information to the team and one tool to pull (retrieve) information is more than enough.
- *Classify*: Make a classification of types of information. For example, information about production deployments should not be combined with information about deployments in test environments.

- *Target*: Broadcasting notifications to all team members isn't such a good idea. A better way is to target the information to a specific person. If a team member pushes code to a specific branch, the result of the build execution should be sent only to that particular person.
- *Combine*: Information is shared as soon as it is available, but in some cases, it makes sense to wait for a while, gather all information during a certain time interval, combine it in a presentable state, and share it with the team. An example of combined information is a release note.
- *Filter*: If information is not used, why bother to burden the team with it? Make sure only the information that makes sense is also shared with the team.
- *Viewpoints*: Sometimes you just want to know what has been deployed to a certain target environment. In other cases, you want to know who broke the build. The information must be presented with different viewpoints in mind. An ALM platform or a specific reporting tool can help with that.

Requirement: Use short feedback loops, but don't overload teams with too much information

Short feedback loops are a core principle of CI/CD, but finding the right balance between providing enough information or too much information is difficult. Teams are informed about successful or broken builds, test results, pending manual actions, etc. The email inbox piles up with emails, and developers tend not to look at them anymore because it is also difficult to make a distinction between urgent emails and informative emails.

Using a good combination of different communication tools, such as email, Microsoft Teams, Discord, and Slack, and a mix of the information strategies described at the beginning of this paragraph, reduces the risk of information overloading.

Requirement: Create feedback loops in every stage.

It may be a no-brainer, but feedback must be given as soon as possible, so instead of waiting until the pipeline is finished, the result of a failed step must be sent to the team as soon as the failure occurs.

Requirement: Automate the creation of release notes.

Release notes are useful because they define clearly what is included in a specific release. However, assembling the information to construct a release note should not be done by hand. Automate this, based on the information of the commits, the pull requests associated with a particular release, and the test results.

Requirement: Provide insight into versions of artifacts installed on test and production environments.

Teams sometimes use multiple fixed test environments on which several test types are executed and on which different versions of artifacts are deployed. Often there is no good insight into which version is installed where. Provide a dashboard containing all environments—including production—and all versions of installed artifacts.

Security (General)

Security plays an important role in developing, implementing, and managing pipelines. The ALM platform or integration server, the related tools, and the pipelines themselves are potential attack surfaces, so they need to be protected and monitored. Don't forget that if applications have to meet certain standards, such as the Sarbanes–Oxley Act (SOX), Health Insurance Portability and Accountability Act (HIPAA), or Payment Card Industry Data Security Standard (PCI DSS), it might be assumed that the

software supply chain also has to comply with these standards. Most of these standards have a component focused on security in the software supply chain.

Here is where the NIST Cybersecurity Framework [13] can play a role. The NIST Cybersecurity Framework is a valuable source helping business organizations to identify risks, protect resources, detect vulnerabilities, and respond to and recover from security incidents. It is an extensive framework and covers various security aspects targeted at people, processes, and technology. Use the framework as guidance to define CI/CD security requirements.

For example, one of the categories in the framework deals with supply chain risk management. Subcategory ID.SC-2 states the following:

> *ID.SC-2: Suppliers and third-party partners of information systems, components, and services are identified, prioritized, and assessed using a cyber-supply chain risk assessment process.*

If this is brought up in the context of external libraries used for building an application, it is made clear that the origin of such a library must be assessed first. Just grabbing some software from the Internet and bringing it into your production environment is not a good idea.

Requirement: Use a vault to store tokens, keys, secrets, and passwords.

Ideally, all secrets—passwords, tokens, keys, credentials—used by the application must be stored in a secure vault. Depending on the exact requirements, this vault may have certain characteristics. It can be a software vault or a Federal Information Processing Standard (FIPS) 140-2 level 3 compliant hardware security module (HSM). The pipeline has to make sure that these secrets are stored in the vault, either by generating them in the vault itself or by securely transferring the secret to the vault. Some ALM platforms are supported by a vault to store secrets.

Requirement: Refine access by setting permissions for a user or group.

Reading or writing actions to an SCM repository, starting a pipeline, and performing a dual control are typical functions that require access control because one person may, for example, start a pipeline, but this person is not allowed to approve their pipeline. This requires a fine-grained access matrix, realized using a role-based identity and access management (IAM) setup.

Requirement: Check for drift detection.

Drift detection checks whether the actual infrastructure is still the same as compared to the infrastructure code. If there is a difference between the infrastructure code and the actual infrastructure, the change was applied manually. Drift detection is often done on a scheduled basis.

Requirement: Perform a vulnerability analysis on third-party libraries.

Log4J versions 2.x until 2.16 contained vulnerability CVE-2021-44228. This was a good example of a vulnerable third-party library. Scanning third-party libraries in the pipeline on vulnerabilities should be a mandatory task. Even if you think you don't even use certain libraries, you may be surprised. Transitive dependencies, in which a third-party library makes use of another third-party library, are common. The third-party library you use may not have vulnerabilities, while the transitive dependency does. The same applies to COTS software used in organizations. Life-cycle management of software ensures that the software does not contain vulnerable third-party libraries, but unfortunately organizations often make use of software, which is beyond end of life or use an old version of the software.

Requirement: Scan third-party libraries on malware or viruses.

Even if a library comes from an authenticated source and the integrity is guaranteed and if the library does not contain any vulnerabilities, then it can still contain malware or viruses. Scanning third-party libraries on malware and viruses is recommended.

Note This does not apply only to third-party libraries used to build an application in the pipeline; it also applies to COTS applications that make use of third-party libraries.

Requirement: Prevent deletion of resources.

Any resource associated with the creation and deployment of a release (candidate) artifact must be prevented from being deleted. From an audit point of view, the resources involved in creating, testing, and deploying an artifact must be protected from deletion. This applies to code (repositories), work items, pull requests, pipeline definitions and pipeline runs, artifacts, testware, etc.

Requirement: Connections between the ALM platform/integration server and external tooling must be secure.

The ALM platform or integration server deploys artifacts to a production environment. This implies that the ALM/integration platform also needs to be a production environment (e.g., running in the same production network segment). Access to the platform must be secured, and connections with other tools must also be secured. This is done using standard solutions, such as an HTTPS connection with mutual TLS (mTLS). Any data passed between the ALM platform/integration server and connected systems is encrypted, and mutual authentication is established.

Requirement: All infrastructure is hardened.

This is related to the previous requirement. If you manage the CI/CD infrastructure yourself, make sure the servers on which the tooling runs are hardened. Hardening your servers reduces the attack service of the infrastructure. If you use a SaaS solution, the provider of the service takes care of hardening the infrastructure.

Requirement: Clean up and secure your CI/CD workspace.

No, this is not about your desk. It applies to the workspace used during the execution of a pipeline on a server. Integration servers make use of a workspace. Often this is a directory on a server. Data is stored in this workspace. This includes code but also secure files, containing information that shouldn't be disclosed. This is, for example, an application property file with database credentials. In the first place, other users and other pipelines should not be able to access your pipeline workspace. Make sure this access is blocked, and the workspace is only allowed to be accessed by the intended pipeline.

After the pipeline is finished, the whole workspace must be wiped clean. Dangling workspaces is a risk that should be avoided. Make sure that the workspace is clean after the pipeline is finished. If not, add a *post-task* to the pipeline, which is always executed. The post-tasks perform the cleanup.

Requirement: Make use of a container-based CI/CD workspace.

As an alternative to storing data on a file system or network-attached storage (NAS), a container-based workspace can be considered. A newly running (Docker) container starts with a clean workspace, and when the container stops, the local data is wiped because data in a container isn't persisted by default.

Requirement: Roll back or roll forward if a deployment goes bad.

A deployment can either fail or not fail but the updated service produces unpredictable or incorrect results. Rolling updates/canary deployment is a way to mitigate the impact and is highly recommended, but it does not prevent deployment failure. In all cases, a reaction is required. You need to do one of the following:

- Roll back to the previous version and fix the damage
- Fix it and roll forward

There is a tendency to say you always need to roll forward, but that depends on the viewpoint. A simple web application with a corrupted layout is something completely different than a payment or trading system with a recovery point objective (RPO) of zero and an RTO of nearly zero. Without judging the situation, one can only conclude that "it depends." What is more important in this context is the fact that it must be possible to perform a rollback or roll forward using a pipeline. A rollback not only means undeploying the new artifact version and redeploying the old version, but it also has to execute rollback scripts to reverse the changes already made in the database, roll back messages in a queue, or roll back any data already propagated to other systems. Also, a roll forward may involve more than just installing a fixed app. Any corrupted data needs to be fixed also.

This is not for the faint of heart, and whatever strategy is used, it requires some thorough thinking up front and needs to become part of your test strategy. Without a proper rollback/roll-forward vision, you will continue to work on your pipeline endlessly. Be prepared for that in the pipeline design.

Requirement: Only deploy artifacts to production with a higher version.

This requirement seems to be contradicting the previous requirements because checking whether the deployment always has a higher version sort of prevents a deployment rollback. That is also not the intention. In most cases, a deployment just succeeds, and the installed version is always the latest one, which has a higher version number than the previously installed version. An additional check on the existing version on production versus the version that is going to be deployed prevents the installation of older versions. This requirement implies that the versioning scheme has an order. Using a commit hash as a version does not work in combination with this requirement. In the case of a rollback, this check should be suppressed, of course.

Requirement: Security tests are automated.

Dynamic Application Security Testing (DAST) includes test techniques that expose security weaknesses and vulnerabilities present in an application. As the word already suggests, DAST tools perform a dynamic test that tries to uncover cross-site scripting, SQL injection, cross-site request forgery, information disclosure, etc. DAST tasks should be integrated into the pipeline as part of the stage in which tests are executed.

In addition to DAST, a penetration test (*pentest*) can be executed. Of course, this depends on the risk appetite you want to prepare to accept. It is a practice performed by cybersecurity professionals trying to identify weaknesses in a system. Pentests are often performed as manual tests. Pentesting as a service (PTaaS) is an emerging technology that helps to fill this gap by automating parts of the work. Considering the current state of cybercrime, PTaaS is an interesting area to consider.

Compliance and Auditability

Although compliance and auditability could be classified under the "Security (General)*" section*, the subjects are too dominant not to see them as a separate requirements analysis area.

Compliance refers to the act of following laws, regulations, guidelines, and specifications that apply to a company or industry. It is the process of ensuring that an organization is adhering to the laws, regulations, and standards that apply to its business.

Auditability is the quality of being capable of being audited or the ability to be examined and verified. In the context of compliance, auditability refers to the ability of an organization to provide evidence that it is complying with laws, regulations, and standards.

Requirement: All changes are traceable.

For auditability reasons, every change in the build and release process must be traced back to each resource—or entity—that was responsible for this change. These resources are in some way linked to each other. For example, if a work item is implemented, it must link to a design, the source code, the code reviews, the build results, etc.

The following resources play a role in application development:

- *Requirement*: A requirement is described in a design and referred to by one or more work items.
- *Design*: A design describes one or more requirements.
- *Work item (epic, story, or task)*: A work item is referred to by a requirement and referred to by a pull request.[2]
- *Commit*: A commit is created by a developer and implements a work item. The commit refers to the application code in the repository. A build refers to the commit, establishing an audit trail.
- *Application code*: The application code is developed by one or more developers and realized by one or more commits. The application code is used by a build to create artifacts.
- *Pull request*: A developer creates a pull request, which is reviewed by another developer. The pull request refers to a work item, so the developer knows which commit was involved and which application code they have to review.

[2] Not all teams use pull requests.

- *Developer*: A developer develops application code, commits the code to the repository, and creates pull requests. The developer also reviews the pull requests of their colleague.
- *Build*: A build uses application code to create artifacts.
- *Artifact*: An artifact is created by a build and deployed by a release. The artifact runs on test and/or production environments.
- *Release*: A release deploys one or more artifacts to test and/or production environments and generates a release note.
- *Release note*: A release note is generated as part of a release.
- *(Prod + test) environment*: A test and production environment runs an artifact, which is deployed by a release.
- *Test run*: A test run is executed on one or more test environments.
- *Test specification*: A test specification covers a requirement and is executed by a test run.

The relations between these resources are visualized in Figure 3-1.

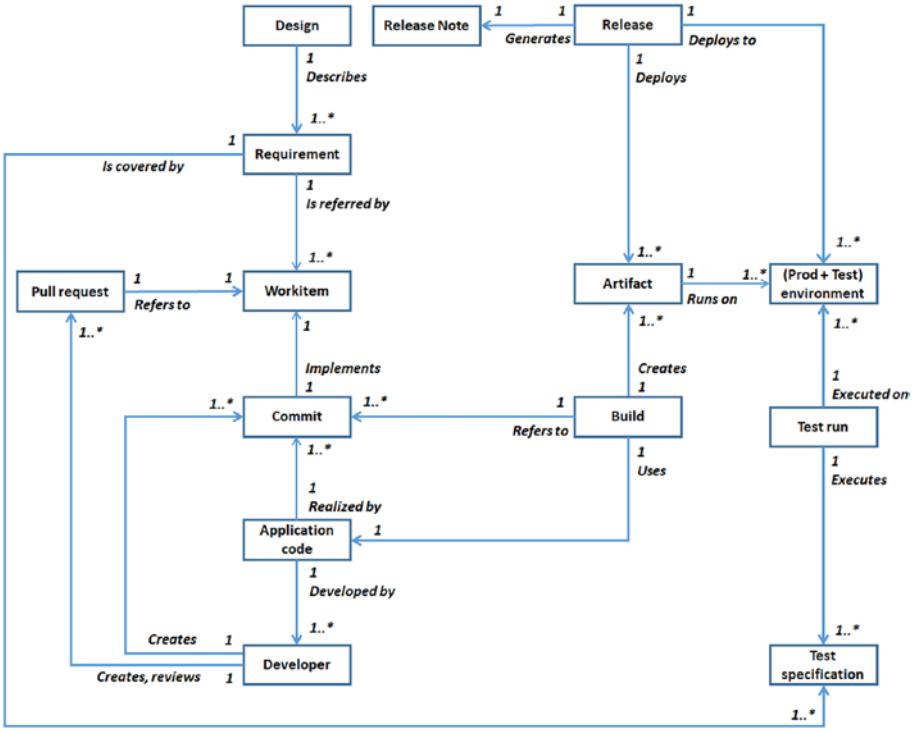


Figure 3-1. *Relations between resources involved in application development*

Each resource in this diagram must be traced back to another resource. Questions an auditor in your organization could ask are "Which artifact version runs in the production environment, and which test runs were executed for this artifact?" or "Which requirements are associated with a specific version of an artifact, and in which application code is this realized?"

Traceability is also very valuable to determine the origin of a failed test. If a test can be traced back to a work item and the associated commit, it becomes easy to pinpoint the exact code that caused the test to fail. This even makes it possible to automatically exclude this code and rebuild the artifact again.

Requirement: Tag everything.

This requirement describes the "how" of the previous requirement.

To make sure that it is clear which version of an application is running in production, which artifacts were deployed, which code was used to build an artifact, which pipeline runs were responsible for building and deploying the artifact, and which test runs were executed, tagging should be applied. In the ideal world, all resources depicted in Figure 3-1 associated with the creation of the application running in production should be tagged. Unfortunately, this can become complicated, but in cases where tagging can be applied, it is recommended to do it. Use a uniform tag to identify a release; a release version is recommended.

Requirement: All code is peer-reviewed.

Code is checked by a colleague before it is merged into the trunk. This ensures quality and prevents unauthorized changes. This requirement does not apply only to application code but also to infrastructure code and pipeline code. Most ALM platforms include options to create a pull request.

Requirement: Only artifacts built by a pipeline are allowed to be deployed to production.

It is important to only allow artifacts that have been built by a pipeline to be deployed to production because this helps to ensure that the artifact being deployed has been properly tested and verified. It also helps to prevent issues such as bugs, security vulnerabilities, or other problems from being introduced into the production environment, thus preventing the production environment from becoming unstable and vulnerable.

This requires special measures that prevent the deployment and installation of artifacts retrieved from untrusted sources. There are several options to guarantee that artifacts are built only by a pipeline.

- The production environment accepts only signed artifacts. Only artifacts signed by the pipeline are accepted, and because the private key used for signing

the artifact is managed by the pipeline, there is no way to create this digital signature in another way. The validation of the signature is done on the target platform itself.

- The binary repository containing the artifacts is accessible only using a pipeline. Manual upload of artifacts to the binary repository is prohibited and prevented. This gives more comfort knowing that the artifact was at least created using a pipeline.
- Some solutions make sure that each stage in the chain is executed as planned and that the artifact is not tampered with in transit. Frameworks like In-toto and Argos Notary make it possible to validate whether all steps in the process have been executed as defined. The framework makes sure that data related to a step has not been tampered with when passed to the next step. The metadata of each step is gathered and used as input to create a digital signature, which guarantees integrity. The whole process is audited by an external system that verifies all steps. For more information, see [22].

Requirement: Deployment to production is allowed only using a pipeline.

In addition to the previous requirement, not only do safeguards guarantee that only artifacts built by a pipeline are deployed to production, but the actual deployment itself must also be restricted, so the deployment can be performed only using a pipeline. Manual deployments must be prevented; otherwise, it is impossible to trace back what exactly has been installed on the target environment.

- Access control must be configured in such a way that only the servers on which CI/CD tools are installed are allowed to connect to the target environment. This can be done by a combination of measures, such as IP whitelisting,[3] setting up a local firewall around the production environment (for example, the firewall capabilities of NSX virtual networking on top of a VMWare ESXi cluster), and establishing mTLS connections.
- Access can be limited even more within an ALM/integration platform or other CI/CD tool. Specific pipelines should have access only to target environments, while other pipelines should not. By using a token or nonpersonal user credentials in combination with IP whitelisting, the pipeline can connect to the target environment. The tokens or credentials are not shared by other pipelines and must be rotated regularly.
- Frameworks like In-toto and Argos Notary (see [22]) provide solutions to guarantee that the deployment was performed as defined.

Requirement: Verify that an artifact is not altered between creation and installation.

Created artifacts may never be changed after creation. This ensures that the artifact deployed to production is the one that was intended and is not changed in any way. It, therefore, does not become subject to misuse.

[3] IP whitelisting is not preferred anymore due to maintenance/error-prone situations, especially in cloud environments. Use it only when there's no other option.

This requirement can be realized by a task in the pipeline. This task signs the artifact and adds a digital signature to it. This signature authenticates the artifact, and it is 100 percent sure that the artifact is created by the pipeline. This implies that the target environment must have a mechanism in place that allows only signed artifacts to be installed. The digital signature ensures that the integrity of the artifact remains throughout all subsequent stages.

An alternative is the use of a hash. The hash of an artifact is created in the pipeline and deployed together with the artifact to the target environment. As part of the installation, the hash of the artifact is generated again on the target environment and compared to the hash that was delivered as part of the deployment. Needless to say, this method is a lot less secure.

Requirement: Verify that an artifact after deployment is still the same.

This requirement is an extension of the previous requirement, "An artifact is not altered between creation and installation," but in this context, it concerns the artifact already installed in the production environment.

A continuous scan of the artifact running in production makes sure that the artifact has not been changed after it was installed in production. The scan continuously validates the integrity of the artifact in production, for example, by checking its signature. AWS Lambda code signing is an example of a mechanism to determine whether the running code has been altered.

Requirement: Use only authenticated external libraries and software.

Third-party libraries must be approved before they can be used. How do you prove that the library does not do something harmful? Maybe it does its job but in the meantime also gathers information and sends it to a server outside the organization. Even in cases in which you think it is the software you intended, a hacker may have updated it and saved it under the same name. This means that the location the software is retrieved

from must be approved (by performing an assessment) and authenticated (as part of downloading the software). The software retrieved from this location must be validated on integrity (either using a hash or better, using a digital signature).

This requirement, combined with the previous requirements, implies a chain of trust, from the external developer creating and publishing a library until the creation of an artifact (using the library), deploying it, and running the artifact in a target environment. Figure 3-2 shows an example of such a chain of trust.

- The developer uploads their signed library.
- The signature is validated on the central library server.
- The organization "trusts" the central library server because they performed a security assessment.
- The CI pipeline retrieves the library from the central server[4] and validates the signature to determine whether it indeed originated from the developer.
- The CI pipeline creates a signed artifact, which is validated by the CD pipeline to determine whether its integrity can be trusted.
- The target environment continuously validates the signature of the artifact to make sure it's still the same artifact running on the target environment.

[4] Maybe not directly, but using a proxy or intermediate repository.

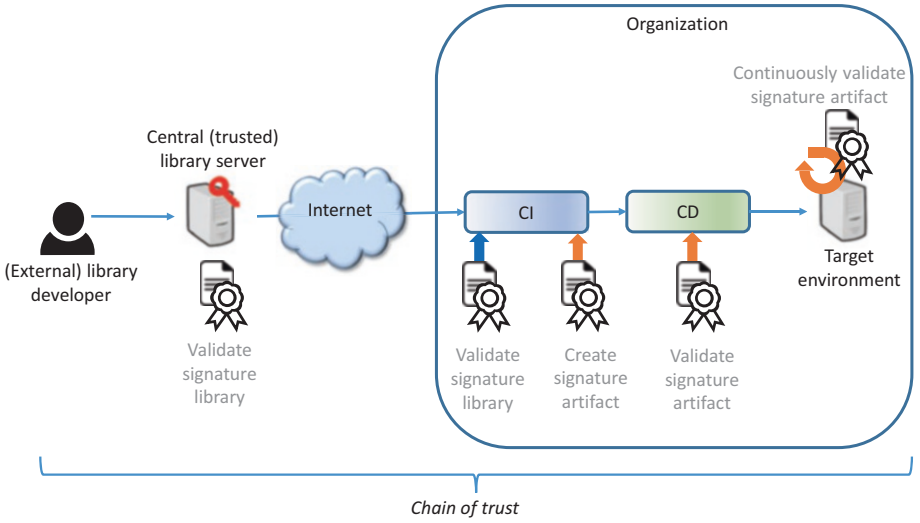


Figure 3-2. *Chain of trust*

Requirement: Resources associated with a release cannot be deleted.

This means that if a release is built and deployed to a production environment, the code in the code repository, the artifact, the work item, the pull requests, and all other related resources may not be deleted. Measures to prevent this have to be taken.

Requirement: Pipelines are scanned for compliance.

Not only applications built by pipelines are subject to code scanning, but the pipelines themselves can be scanned for compliance. A big organization with a lot of DevOps teams might impose certain restrictions or criteria to which a pipeline must adhere. For example, analyzing application code may be mandatory, which means that the pipeline must include tasks to scan the application code. Another example is the availability of a dual control task, which is executed before an artifact is deployed to production.

Requirement: Test data is anonymous.

Various reasons mandate that personal data (personally identifiable information [PII] data) in tests cannot be traced back to a real person. The GDPR rule in Europe is very strict concerning PII data. Data is allowed to be made visible only on a need-to-know basis, and test engineers and developers are usually not allowed to have access to this data. In addition, the use of production data in a test environment also hurts the reputation of the company when this becomes known. The following are measures:

- *Use synthetic data*: This is generated data as an alternative to real-world data. Using synthetic data is preferred over anonymized data because you don't have to touch production data.
- *Anonymize the data*: Data anonymization, also known as *data obfuscation* or *data masking*, involves removing personally identifiable information or altering (production) data so it cannot be traced back to a person. Use anonymized data only if you can't use synthetic data.

Requirement: Pipeline logs may not contain PII data and secrets.

Derived from the previous requirement, PII data may not be used at all in a CI/CD pipeline. And even if PII data is needed in the pipeline, for example, to fill a database table in production, the data needs to be protected.

Various options are possible to protect the data. The simplest solution is to store it as a file, secured from reading by other users than the pipeline. Even better is to encrypt the file. The pipeline decrypts the file as soon as it is needed to fill the table. The decryption key must also be stored in a secure location within the pipeline, of course. Another alternative is to store the file in a vault.

Resource Constraints

Resource constraints affect the pipeline negatively and introduce queuing, very long execution times, or even a complete halt of the whole ALM platform/integration server. The underlying reasons are often a lack of computing (CPU) resources, insufficient disk space, and network congestion. These usually occur when the pipelines are already put into use. It seems like these problems suddenly happen to you and you have to deal with them as soon as they happen, but that's very short-sighted.

As soon as you start with the design and development of your pipelines, you should have some idea about the number of apps, the number of pipelines, and how many pipeline runs are expected. The sizing of the CI/CD infrastructure is an educated guess, which should at least give enough confidence that the pipelines can do their work given all requirements. In addition, some optimizations can be done.

Requirement: Parallelize code analysis scans.

If code analysis consists of multiple scan types, it may take a long time to complete if all tasks are executed sequentially. A solution is to parallelize these tasks. It is good practice to include this already in the design because the different types of code analysis scans do not have any relation to one another.

Requirement: Parallelize tests.

Not only can code analysis scans take a long time, but especially test runs are prone to take a long time. Solutions are to execute multiple types of tests in parallel or parallelize tests of the same type. In the case of the latter, tests are divided into small groups, and the groups are executed in parallel. Other approaches are to group tests based on historic timing data and combine the tests in such a way that the test time of each group is (almost) the same.

Requirement: Offload build, source code analysis, and test execution.

Running builds, source code analysis (SCA) tasks, and executing tests on the same server is not recommended. Pipeline runs get queued because the server on which everything runs reaches its limits. Running offloaded builds (e.g., adding nodes to the Jenkins master), running SCA tasks on dedicated servers, and offloading test runs from the build server to a test server are good practices. Even if the pipeline runs on an ALM SaaS solution, this practice still holds in case the servers or agents on which the pipelines run are shared with other DevOps teams in the company. Heavy processing may affect other teams because the agent pool has run out of servers.

Manageability

Manageability is about organizing your pipelines in such a way that changes are easy to apply and your code is not redundant and scattered all over the place.

Requirement: Keep your pipeline code manageable.

Similar to software development, pipeline development can become complex. Sometimes this cannot be prevented, but that's all the more reason to keep development under control. Your pipeline becomes unmanageable if every hobbyist is given the space to add another hobby script of their preference. Using technical standards, naming conventions, and development guidelines is the only way to keep pipeline development manageable.

Requirement: Build once, run anywhere.

"Build once, run anywhere" is a statement originated from the Java and Docker/container world, which also applies to the context of pipelines. An application artifact must always be built once using a pipeline, and the same artifact must be installed in all target environments, both test and production. Environment-specific properties are deployed as part of the application deployment.

Requirement: Store binaries (artifacts and dependencies) in an artifact repository.

In addition to the principle "Maintain a *single source repository*" (store your code—including pipeline code—in a source control management system (SCM)), it is good practice to store all artifacts and dependencies in an immutable binary repository.

Requirement: Do not retrieve libraries or external resources directly from an Internet location.

Instead of directly retrieving a library from an authenticated location on the Internet, it is good practice to store the libraries locally (on-premises or in your cloud account). Pipelines use the local repository instead. In addition to security-related issues, pipelines shouldn't be dependent on the availability of the external—Internet—location. Chapter 5 describes a few options for how to deal with this.

Requirement: Pipeline code is treated as software.

This was already explained and a basic principle of CI/CD. Pipeline code, automation and orchestration code, scripts, and pipeline designs are all stored in a source code management system (e.g., Git), so they are versioned.

Requirement: Fix variables.

In specific cases, it may be needed to use *constant* variables in a CI/CD pipeline or prevent them from being changed at the start or during a pipeline run because this can lead to unpredictable and potentially harmful consequences. You can ensure that the pipeline is reliable and consistent by fixing the variables in a CI/CD pipeline.

For example, if a variable that defines the version of a dependency is changed during a build process, it could cause the build to fail or produce an incorrect result.

Requirement: Use one deployment script for all environments.

Not only the same artifact is used throughout all target environments, but the core deployment scripts must also be the same, which means that the script to deploy an artifact to a system test environment is also used to

deploy to a production environment. The differences between each target environment are parameterized.

Requirement: Use one provisioning script for all environments.

Similar to the previous requirement but referring to the infrastructure code. The infrastructure code used to create a target environment is the same for all target environments. The differences between each target environment are parameterized.

Note The difference between provisioning infrastructure and deployment of an application begins to fade in the context of cloud development. Often, both application code and infra code are combined into one repository, and deployment involves both the deployment of an application and the deployment of the infrastructure.

Requirement: Use a release versioning schema that makes sense.

Sometimes people propose to use a Git tag as a release version, but, to be honest, do you really want an artifact with the name `application-4fbed257bc4b94a4a042a6e38440a0d2b95c16ac.jar`? And how do you communicate with your business about it? "Hey Jim, we just released version 4fbed257bc4b94a4a042a6e38440a0d2b95c16ac."

Use a versioning schema that makes sense and meets the following criteria:

- You must be able to communicate about it.
- It must be generated. It is not continuous if you have to provide the release version to the pipeline yourself every time.
- It must have an order. If you compare release versions, it must be clear which one was the oldest.

Semantic versioning (`major.minor.patch`) is still used a lot as a versioning schema. It is useful when communicating with consumers of your application about whether a new version is backward compatible or contains breaking changes. However, it is a bit tricky to generate. A tool like *semantic-release* can help you with this. Having said that, one might question whether this form of versioning is still relevant today. What is the meaning of a major release if the release frequency is once a day?

Another versioning schema is date-based versioning with a sequence. The format is similar to yyyyMMdd.<seq>, for example, 20230214.3.

Requirement: Pipeline stages and tasks are orchestrated by the appropriate tool.

An ALM platform or integration server is at its core an orchestration tool that executes specific tasks. These tasks may use features, which are added to the platform. These features are already integrated, added as plugins or marketplace solutions, or are manually installed on the platform. But in some cases very specific tooling is required. One category is, for example, tooling used for deployments. Perhaps it is possible to develop a deployment tool yourself, but often there are better solutions available, preferably solutions complying with an open architecture. Also, make sure where this tool is installed. A deployment tool is sometimes installed on the ALM platform/integration server itself (e.g., Cloud Foundry CLI), it is installed on the target environment (e.g., AWS CodeDeploy), or it can be a stand-alone deployment tool on a separate server.

Operations

Operations tasks must be automated as much as possible. Using a pipeline to orchestrate these tasks is a logical choice.

Requirement: Automate operations tasks.

Pipelines are not only used for provisioning infrastructures or building and deploying application artifacts. Also, one-off operations tasks should be automated using a pipeline. Here are some examples:

- Renewal of certificates
- Inserting data into a configuration table
- Creating asymmetric key pairs
- Onboarding new clients

Make sure the operations pipelines and associated scripts are versioned in an SCM.

Requirement: Integration infrastructure requires an SLA and a business continuity/disaster recovery plan.

Business continuity should not be limited to the business application itself; it also applies to the software supply chain. Not being able to release an application may also hurt business continuity. What are the alternatives to deploy an application in the case of a disaster of the CI/CD platform? The service must be restored, or an alternative must be considered for the time being.

So, an SLA must be defined for the pipeline, and based on this SLA, a business continuity/disaster recovery plan must be created and regularly tested.

Requirement: Patch the integration infrastructure regularly.

An integration infrastructure must be treated the same way as a production environment on which an application runs. An unpatched integration infrastructure is vulnerable, and the latest patches have to be applied on the servers of the ALM platform and the servers on which other CI/CD-related tooling is installed.

Quality Assurance

Quality assurance (QA) involves source code analysis, both static and dynamic, and testing. Testing is meant here in the broadest sense of the word. It not only involves various testing types but also the creation and management of test data and testware. In addition, security testing is considered part of QA, although security is treated as a separate topic.

Requirement: Application code must be scanned on code quality.

Application code must meet a certain code quality. Static code scanning is performed on the application code to validate bugs, coding standards, complexity, bugs, nonperforming code, etc. The code must also be checked for—security—vulnerabilities. Scanning code provides confidence that the code quality of the application code is sufficient.

Dynamic scanning is validating the application in the runtime environment to determine whether it contains security vulnerabilities (e.g., using automated fuzzing).

Requirement: Infrastructure code must be scanned on code quality.

In addition to scanning application code, also infrastructure code must be scanned on code quality. This involves both static scanning of the infrastructure as code (IaC) and dynamic scanning of the target environment.

Static scanning involves validating infrastructure code such as AWS CloudFormation and Azure ARM templates. Dynamic scanning involves validating whether an infrastructure resource in the target environment is not misconfigured.

Both types of scanning complement each other, but considering the "shift-left" principle, most of the issues and misconfigurations should preferably be detected by static IaC scanning.

Requirement: Pipeline code must be scanned on code quality.

Because most pipelines are developed as code, they also need to meet a certain code quality. Although scanning the pipeline code in the pipeline itself is an option, it is a bit odd. That would be a bit like a fox guarding

the henhouse. Scanning the pipeline code should preferably be done by another entity, outside the pipeline.

Requirement: Pipelines are testable.

A realized pipeline must behave as it was intended. This means that a thorough test is needed in which it is possible to execute test runs with the given pipeline code, without actually endangering normal pipeline behavior or deploying artifacts by accident. The pipeline must be functionally tested, but also nonfunctional aspects must be tested. Is the performance of the pipeline sufficient?

Requirement: Use quality gates.

Some ALM platforms introduce the concepts of approval and gate. An *approval* usually refers to a manual task. A *gate*, sometimes called *quality gate*, often refers to an automated approval. A quality gate is a milestone where the outcome of a pipeline stage is validated to see if it meets the necessary criteria to move into the next stage. Here are some examples:

- *Approve pull request*: Before code is merged back into the main branch, it needs to be approved by colleagues. Approval is often done using a pull request. This is a manual validation. Code that is not reviewed using a pull request cannot be merged with the mainline.
- *Analyzing code quality*: This means that the application code must meet certain quality criteria. Code scanning tools must have integrated policies in which thresholds are defined. Code that exceeds the threshold is considered "acceptable." Code that does not reach the threshold is of poor quality. The pipeline should fail in these cases. Examples of such thresholds are as follows:
    - Unit test coverage must be higher than 80 percent.

    - Code may not contain vulnerabilities with a Common Vulnerability Scoring System (CVSS) score of 7 or more.
    - Code may not contain Blocking or Critical issues.
    - Code or property files may not contain passwords, tokens, or any other secrets.
- *Integrity check on artifact*: This refers to performing an integrity check on an artifact before it is deployed to production. The artifact must have a valid digital signature. If not, the deployment cannot proceed.
- *Validate test results*: Not only tests are automated, but also the validation of tests can be automated. In principle, all automated tests must pass; otherwise, the pipeline stops. Release candidates must be earmarked with the test result to prevent a release candidate is deployed to production for which not all tests passed or testing was incomplete.
- *Validation of the main branch*: Only artifacts built from the main branch are allowed to be deployed to production. This approval must be automated. Artifacts originating from other branches are not allowed to be deployed to production.
- *Validation of the artifact version*: The version of the artifact must be higher than the version of the artifact in production. The artifact cannot be deployed if the version is lower.

Requirement: Define entry and exit criteria.

As already explained in previous chapters, validation of entry criteria means that the pipeline starts with the correct starting situation.

Arguments are passed from an external system to the pipeline, which determines whether it can start with the given data. Validating the exit criteria means that all preconditions to deploy and install the software in the target environment are met. Here are examples of entry and exit criteria:

- *Entry criterion*: Committed code must be associated with a pull request.
- *Entry criterion*: Mandatory variables used in the pipeline are configured.
- *Exit criterion*: Only signed artifacts may be deployed to production.
- *Exit criterion*: Verify that an artifact deployed to production is indeed a release candidate and not a snapshot build. Deployment to production is possible only with a release artifact of which the code originates from the main branch.

More examples are given in Chapter 4.

Requirement: Tests are reusable (for next test cycles/regression).

When running an (automated) test, the starting position must always be the same to compare different test runs. Starting with the same test environment, the same initial test data and the same test framework are key. A "reset" task must therefore be performed before the actual tests are executed.

Requirement: Tests, test data, stubs, and test reports are versioned (e.g., in Git).

Similar to pipeline code, all tests, test data, and stubs must also be treated as software. The set of test resources must be stable and versioned and therefore be stored in an SCM system.

Requirement: Development tools, test code, and testware may not be deployed to production.

The pipeline must ensure/have facilities that development tools (such as compilers), test code, and testware cannot be deployed to production where they could potentially cause issues or expose sensitive information. They are not intended for use in live (production) systems.

Metrics

Metrics are used to assess the state and performance of the teams and the pipelines.

Define key performance indicators (KPIs) that make sense.

The software supply chain is successful if all PKIs are considered successful, but defining these KPIs is not easy. When is the software supply chain considered successful? Of course, this differs for different business organizations.

KPIs are often defined in business terms that contain words like *efficient*, *cost-effective*, *fast time to market*, *high change success rate*, *compliance*, etc. However, a good KPI must also be specific, measurable, achievable, relevant, and time-bound (SMART). So instead of stating that "the pipelines must be cost-effective," it is better to define the KPI as "Costs of the pipelines per month." The trend of the KPI reveals whether the costs go up or down, and it is up to the squad or business representative to determine whether this trend is acceptable.

It is not always possible to find the right metrics in the CI/CD setup that contribute to a KPI. In the case of the KPI "Costs of the pipelines per month," you need to get insight into the actual costs of the ALM platform or integration server. If the ALM platform is a SaaS solution or if an integration server runs on the infrastructure of a cloud service provider, it is easy to get insight into the costs of the resources used. Assume the CI/CD setup consists of AWS CodeCommit, CodeBuild, and CodeDeploy,

all orchestrated using AWS CodePipeline. With 30 pipelines and 1,000 builds per month, the costs are roughly $50 per month. The actual use of this setup may differ. If the number of pipelines or the number of builds increases in time, the trend of this KPI goes up.

A practical approach to defining a KPI is to investigate first what actually can be measured in the CI/CD setup. Define a sensible subset of metrics. These metrics form a good starting point for the definition of useful KPIs.

The following are some examples of KPIs in the context of CI/CD. Note that some KPIs are more related to DevOps and not specifically to CI/CD, for example, *meantime to repair/recover* and *mean time between failures*. That is the reason why they are not included in the following list. Of course, the list is not exhaustive (and you don't have to implement them all).

- *Execution time of each stage in the pipeline*: This KPI says something about the speed of build and deployment. If a stage takes a lot of time, search for the bottleneck. Is the code analysis stage taking a long time? Determine whether it is possible to execute the different code analysis tools in parallel instead of running them in sequence.
- *Queued pipeline distribution per day*: If queuing takes place during the day, spot how big this queue is. Are all Jenkins executors occupied (in case you use Jenkins)? Increase the number of executors or add a slave build server. Note that this is not necessarily a KPI, but more a technical metric.
- *Test success rate*: Ideally, the number of failed automated tests—executed by the pipeline—must be zero; in other words, the percentage of passed tests must be 100 percent. As soon as a test fails, an investigation is needed, which also costs time.

Checking the success rate of the tests says something about the amount of time spent on the investigation of failed tests.

- *The number of failed builds per day*: Spot the trend. Is there an accumulation of failed builds? Then take a look at what could be the cause.
- *Costs of the CI/CD pipelines per month*: Running pipelines costs money. This KPI gives insight into the costs of using the ALM platform/integration server.
- *Availability of the ALM platform/integration server per month*: If the ALM platform/integration server has availability issues, it reflects on the capability to deploy an app. If a hotfix is needed and the integration infrastructure is not available, this means business impact. Having insights into the availability of the platform makes sense in the long run.
- *The number of production deployments per month*: The number of deployments to production directly gives insight into the capability of the team to deliver fast (or not).
- *The number of work items closed per sprint*: Closing a work item does not always result in a production deployment, so a low number of production deployments per month does not necessarily mean that the team cannot deliver. Some work items are bundled into one production deployment, so knowing the number of closed work items in a sprint combined with the number of deployments in a sprint gives even more insight into the delivery capabilities of the team.

- *Change lead time*: Another way to look at the team's delivery capability is to determine the *change lead time*. This can be measured by looking at the time between the first code commit and the deployment of the code to production. Please note that this is not the same as lead time, cycle time, and mean time to change (MTTC).
    - *Lead time* is measured from the moment a work item was created until the code was deployed to production. This KPI is useless. There may be good reasons to create the work item so far in advance.
    - *Mean time to change* covers the *lead time*, but also includes the business analysis and design phases. The MTTC is even more difficult to measure because these metrics are often not registered or easily available.
    - *Cycle time* defines the moment a work item in the issue tracker is accepted until the code is deployed to production. It is the only KPI of the trio that has real value.
    - *The number of dirty and orphan commits per month*: A dirty commit is a commit with an invalid work item ID in the commit message. An orphan commit is a commit without a work item ID in the commit message. These KPIs can be used to identify whether the teams' workflow hygiene is in good order.

Monitoring

Monitoring is the process to collect data to identify, measure, validate, visualize, and alert about the following:

- Availability
- Resource use/capacity
- Performance
- Security breaches
- Events (expiration events, pipeline events, system events, and so on)

Monitoring tools generate alerts to anomaly events and help developers to solve issues. Monitoring tools used to monitor CI/CD pipelines should be flexible enough to do the following:

- Monitor KPIs of the CI/CD process
- Visualize trends on a (custom) dashboard
- Define KPI tracking by setting upper and lower thresholds
- Alert in case a KPI trend goes up or down a predefined threshold
- Perform system monitoring on the ALM platform/integration infrastructure, validating CPU usage, storage usage, etc.

Requirement: Monitor KPIs.

Defining a KPI is one thing; retrieving the metrics, making it visual, and monitoring the KPI is another. The monitoring tool must be flexible enough to visualize KPIs with custom timeframes, such as a month, week, or day. Thresholds are defined to determine whether a KPI reaches a

critical or unwanted state. A combination of a time-series database and interactive analytics and monitoring tool is perfect for this case. Popular is the combination of InfluxDB and Grafana.

Requirement: Monitoring must be continuous.

A system that monitors the pipelines must always give the current status of the situation. Immediate feedback also applies to monitoring. Information feedback is done both on a pull basis, using custom dashboards to visualize the KPIs, and on a push basis by generating alerts that actively inform the team about a trend breach.

Requirement: Pipelines to manage infrastructure components life cycle.

Infrastructure is not a static configuration. It consists of various components with a certain life cycle; PaaS/IaaS services that need patching, or secrets that require rotation. One concrete example is certificate management.

Certificate management is often laborious if not fully automated. Some systems make use of a large number of certificates, which expire at different times. The team must have clear insight into when each certificate expires, so this should be automated. A scheduled pipeline can check which certificates will expire soon. In addition to notifying the team about the expiration, the pipeline can even automatically request and install a new certificate.

Requirement: The ALM platform/integration server is monitored.

Monitoring pipelines involves using metrics to determine KPIs and determine whether the execution of the pipelines is still in good order. However, the infrastructure on which the pipelines run should also be monitored. Checking whether CPU usage is still good, whether there is sufficient disk space, or validating whether the connection with external systems is still up are typical aspects to monitor. A tool such as Splunk can be used to monitor the integration infrastructure.

Sustainability

Sustainable computing is an emerging trend that focuses on reducing the carbon footprint generated by the information technology industry. To put things into perspective, the annual energy consumption of the global Bitcoin network as of today is roughly 142 TWh, according to the University of Cambridge (see [1]). That's about the size of the electric energy consumption per year of the whole of New York State. These are dazzling numbers. And not only the carbon dioxide footprint of the Bitcoin network is huge, but also trends like AI, Big Data, and other compute-intensive processes have a big impact on the environment.

Sustainable computing becomes an important factor in architecting, designing, implementing, and operating IT systems. This includes continuous integration and continuous delivery pipelines.

Requirement: Define sustainability goals.

> *"Sustainability isn't one optimization; it's thousands."*
>
> Reference [29]

It is important to optimize pipeline processing in such a way that the carbon dioxide footprint is low but the required functionality is still provided. The team has to realize that, for instance, older hardware and underutilized server capacity are not optimal for energy consumption, and executing one unnecessary pipeline run is one too many. It is recommended to add a sustainability requirement because sustainable computing is here to stay.

Governance

Governance involves managing the organization and teams in their CI/CD journey.

Requirement: Involve the entire team in CI/CD implementation.

Do not fall into the "trap" to assign only one or two people to be responsible for CI/CD. Instead, encourage all squad members to contribute something. Let everybody from the squad pick up a small user story to get them motivated to continuous contributions to the pipeline.

Requirement: Measure the team on CI/CD maturity.

Different DevOps teams have different levels of maturity when it comes to CI/CD. A continuous delivery maturity model helps in identifying how a team scores on various topics. There are many models available that can be used as input to measure a team's CI/CD maturity. This kind of assessment is often in the form of a questionnaire. It is a good practice to assess teams every year.

Requirement: Determine what maturity level is most appropriate.

Teams starting with CI/CD and pipeline development must define their ambition. A continuous delivery maturity model can also help in identifying the level of maturity the teams want to aim for. Perform this exercise at the start and create a road map containing the CI/CD milestones.

Requirement: Measure CI/CD in the business organization.

Sometimes, a company is organized in such a way that it has an inhibitory effect on CI/CD. Procedures and supporting departments are not yet ready for CI/CD, or DevOps teams are not on par with a certain CI/CD ambition. Assessing teams and the organization as a whole helps in getting insight into the CI/CD maturity level of the organization. This assessment should be performed periodically to validate the change in maturity, which helps to make adjustments in the CI/CD migration process.

Summary

You learned about the following topics in this chapter:

- A requirements analysis forms the basis of a good pipeline design and realization.
- Pipeline requirements originate from various sources. You have to make sure that no source is overlooked.
- Requirements cover various areas. This chapter included a large list of potential requirements you can use in practice, grouped by area.
- Be inspired by the list of requirements discussed. Even if you don't use them now, you may implement them in the future.

CHAPTER 4

Pipeline Design

This chapter covers the following:

- Why a pipeline design is useful.
- Basic BPMN 2.0 concepts to model a pipeline flow, with a short BPMN introduction.
- The Generic CI/CD Pipeline, a blueprint containing the stages a pipeline should consist of.
- The different stages of the Generic CI/CD Pipeline and their purpose.
- Design strategies concerning branching, build, test, deployment, and release, and how certain aspects of these strategies affect the design of the pipeline.
- Why certain aspects influence the pipeline design.
- Branching strategies, like trunk-based, feature branch workflow, and Gitflow.
- The process of building an application, which involves more than executing a command. Scaling, full builds versus incremental builds, parallel builds, pipeline caching, build targets, cross-platform builds, and multiteam builds will all be covered in this chapter.

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_4

- Aspects related to testing, including the influence of manual testing on pipelines and the execution order of certain test types.
- The different deployment strategies such as re-create, blue/green, rollover update/canary, and A/B testing.
- The types of release strategies and their differences.
- Other considerations that may affect the pipeline design. Examples are separation of concerns, resource constraints, and commercial off-the-shelf software.

Design

A pipeline design is a specification of how to construct a pipeline. It describes the following:

- The CI/CD process in general, the pipeline stages that make up the process, and the individual tasks within a stage. It describes the process in words and visualizes the activities that take place within a pipeline.
- The flow of the pipeline. The conditions that shape the process flow act as gateways, allowing the pipeline to continue or halt until a certain condition is met. These gateways also determine possible alternative paths in the flow.
- The interaction with surrounding systems.
- Input from external systems needed to execute activities in the pipeline, for example, a trigger from an external system to start the pipeline.
- Output from the pipeline to external systems to delegate activities to these systems.

In addition, a design helps with understanding the software delivery process's behavior and helps structure the pipeline code during development. If there are a lot of common cases, certain design patterns emerge. These design patterns provide a good starting point for the design.

Tip A pipeline design is not an extensive book or report, detailed to the extreme. Its purpose is to understand the problem domain and support the realization of a pipeline. It is often used as a discussion document in the team, so try to keep it modest in size. It is important to realize that a design is a means of eliminating bad decisions. The rest is a matter of taste.

CI/CD and Pipeline Design Approach

A pipeline design describes the orchestration of a process or workflow and has a lot of similarities with modeling business processes. So, the question is whether the business process modeling paradigm can also be used as the basis for a pipeline design. The answer is yes, and one method to visualize the process and its stages is to use BPMN notation (see [2] and [16]).

BPMN 2.0

Where the requirements analysis phase helps you understand the problem domain, the BPMN diagrams help you understand the software delivery process flow, the individual stages and tasks in the process, and the interaction with other systems. The notation used in this chapter is BPMN 2.0.

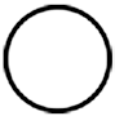
BPMN 2.0 uses a certain notation with specific icons, called *elements*. The set of BPMN 2.0 elements is limited, and because the pipeline flows are not very complex, a subset of these elements is used throughout this book.

A remark to the BPMN purists out there. You will probably detect possible improvements in the models. I would like to know that, of course, but as long as a model describes the essence of the flow, it serves its purpose. A summary of the most used elements and some basic BPMN examples are presented in the next paragraphs.

BPMN Elements Overview

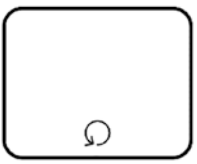
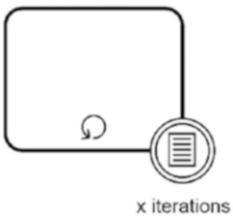
BPMN uses various elements—icons—to model business flows. Table 4-1 presents an overview of the most used ones.

Table 4-1. *BPMN Elements*

BPMN Element	BPMN Name	Description
	Start event	Starts the flow. The use of a start event in a BPMN model is optional.
	End event	Ends the flow. The use of an end event in a BPMN model is optional.
	Error end event	Ends the flow with an error.
	Message intermediate catch event	Acts as a trigger to start a task. It is used for example to identify the trigger that starts the pipeline.

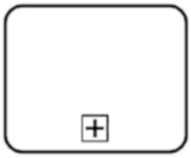
(*continued*)

Table 4-1. (*continued*)

BPMN Element	BPMN Name	Description
	Timer intermediate catch event	Acts as a time trigger to start a task. It is used for example to start a pipeline based on a schedule.
	Task	A task is the smallest execution unit in a pipeline flow. A stage (subprocess) consists of one or more tasks.
	Manual task	A task performed by a user, but without making use of an ICT system. Searching for an order in a drawer is an example.
	User task	A task performed by a user, making use of an ICT system. An example is a dual control task right before an artifact is deployed to production.
	Task with looping marker	Indicates that the task is repeated.
	Repeating task with intermediate conditional event	Repeating tasks with a condition, for example, “This task iterates three times.” Use this construction to make clear how often the task is repeated.

(*continued*)

Table 4-1. (*continued*)

BPMN Element	BPMN Name	Description
	Subprocess collapsed	A subprocess contains other flows, pipelines, and stages and is used to simplify a workflow design. Similar to a regular task, a subprocess can also be repeated using a looping marker.
	Subprocess expanded	The subprocess, but then expanded, so its content is visible.
	Pool (with one lane)	This book mainly uses pools to identify a system or an actor. This can be Git, an email server, a wiki, or an issue tracker, but it can also represent an ALM platform, an integration server, or a software delivery pipeline, depending on the context of the diagram.
	Pool (with two lanes)	Lanes are used to identify units in the pool. For example, the pool identifies a pipeline while the two lanes identify two stages in the pipeline.

(*continued*)

Table 4-1. (*continued*)

BPMN Element	BPMN Name	Description
	Exclusive gateway	Also called an XOR gateway. The exclusive gateway splits the flow into several other paths based on a condition. Only one of the paths is executed.
	Parallel gateway	Also called an AND gateway. The parallel gateway splits the flow into several other paths. All other paths are executed.
	Comment	Comments associated with one of the BPMN icons.

BPMN in Action

A workflow usually has a begin and an end element. In BPMN terminology these are called *events*. Between these events, one or more tasks are executed. This can be an automated or a manual task. A simple BPMN model with two tasks looks like Figure 4-1.

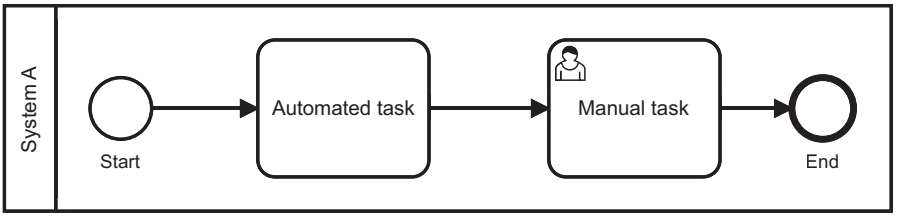


Figure 4-1. *BPMN, example 1*

Figure 4-1 visualizes system A as a BPMN pool. The pool contains two tasks enclosed between a start event and an end event. The start and end events are optional. If the number of tasks becomes very large, they can be clubbed together into a subprocess. To make BPMN diagrams more readable, this subprocess can be collapsed, hiding all underlying tasks, as Figure 4-2 shows.

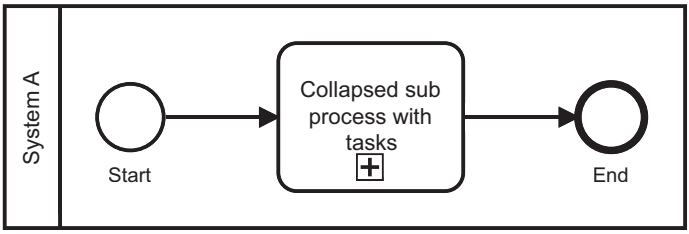


Figure 4-2. *BPMN, example 2*

Tasks can be executed in parallel, or based on certain conditions, alternative paths can be followed. Gateways are a way to model this. Figure 4-3 shows the use of two types of gateways: parallel and exclusive gateways. The model shows that the automated and manual tasks are executed in parallel. The parallel gateway element is positioned both before and after the tasks. The first parallel gateway indicates that both tasks are executed in parallel. The second parallel gateway acts as a converging gateway, meaning that the process continues if both parallel tasks are executed. In addition, the model includes a "happy flow" and an "error flow." The result of both parallel tasks is determined, and based on this condition, the subsequent path either leads to a successful state or ends in an error state. This condition is depicted as an exclusive gateway.

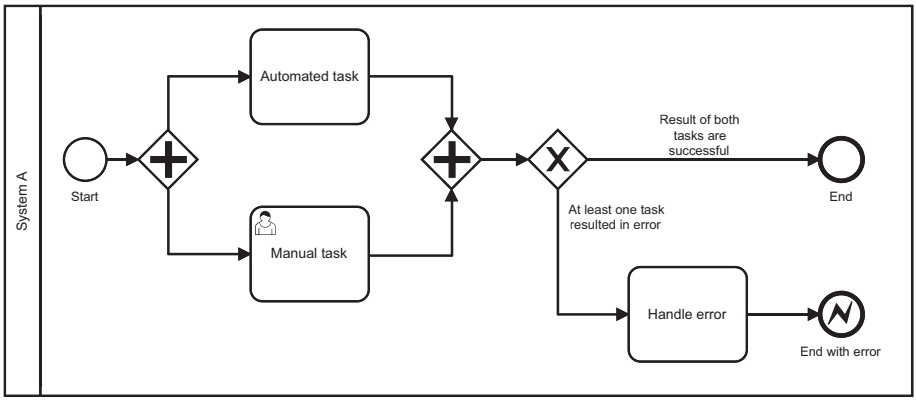


Figure 4-3. *BPMN, example 3*

Figure 4-4 adds a bit more complexity to the model. The *Handle error* task in system A means that previous changes in system B must be undone. System B has two subsystems called B.1 and B.2, and they both must be reset to revert all changes. The two subsystems of system B are depicted as lanes. To inform system B about the fact that the reset must be performed, the model makes use of an event. The event in the model is a message intermediate catch event, indicating that the task *Perform reset* in subsystem B.1 can receive and process this event. After subsystem B.1 has been reset, it calls subsystem B.2 to reset.

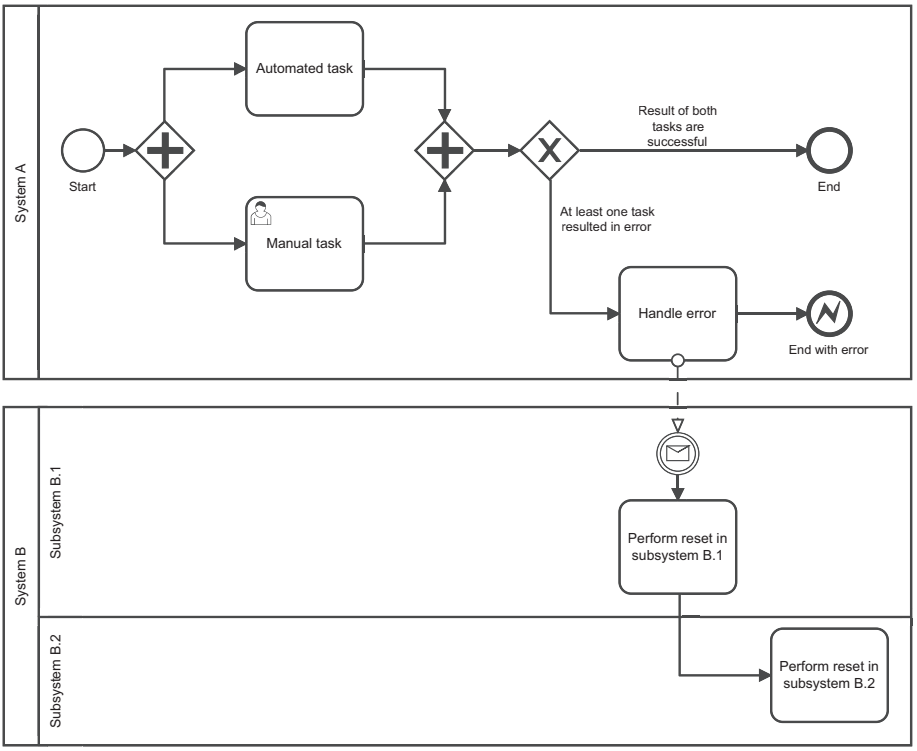


Figure 4-4. *BPMN, example 4*

It doesn't become more complicated than this (at least for the scope of this book). This makes BPMN a good way to describe the workflow of a pipeline flow and helps with the thinking process required to design pipelines.

Level of Detail

A BPMN diagram describes a certain context, which effectively refers to a certain level of detail. There are multiple levels to distinguish.

- Global level, to understand the overall process flow.
- Detailed level, to understand the more detailed tasks.

- The flow. This applies to the conditions that influence the flow. A condition may result in a split, in which multiple tasks are executed in parallel, a condition that defines which path must be followed, or the aggregation of output from multiple tasks.

It is possible to model all levels and the complete workflow in one big BPMN model, but often readability is improved when the global and detail models are separated. This is a matter of taste, of course.

Logical Design vs. Realization

The BPMN models used in this book represent logical designs. In most cases, it does not contain any implementation details because there are thousands of different CI/CD setups, so it is better to avoid implementation details. Creating these models also means, for example, that on a logical level two pipelines may be modeled, while technically, the whole flow can be realized by just one pipeline. In addition, stages and tasks are used throughout the logical representation of the pipeline. The implementation of a pipeline, however, may consist of stages, jobs, steps, and/or tasks, depending on the platform. The developer has to translate the logical design into the technical implementation.

The Generic CI/CD Pipeline

The Generic CI/CD Pipeline is the basic blueprint used throughout this book. It consists of stages, each with a certain purpose. These stages are deliberately kept abstract because a stage in itself can be decomposed again into several tasks. These tasks are completely different in another context, such as the use of the infrastructure, tools, test environment, security, or other constraints.

The stages in the Generic CI/CD Pipeline are positioned in sequential order. If the implementation in sequential order still guarantees the requirements—for example, the requirement of fast feedback—there is no urgent need to restructure the design and the realization of the pipeline, but if these requirements cannot be met, consider parallelization and/or combining some of the stages. See Figure 4-5.

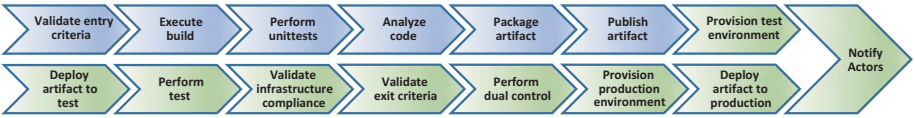


Figure 4-5. *The Generic CI/CD Pipeline*

Mapping the Generic CI/CD Pipeline to a BMPM diagram is relatively easy, but remember that it is still a generic workflow. Subsequent chapters explain the stages of this workflow model in more detail, often in a specific context. Stages in the Generic CI/CD Pipeline are modeled in BPMN as a subprocess because each stage consists of [0..n] tasks.[1] See Figure 4-6.

[1] Some stages are implemented differently, which means that tasks move to a different stage, and the stage ends up with zero tasks, in other words, the [0..n] range. In other contexts, some stages are not applicable, meaning that the stage has zero tasks and is therefore not implemented.

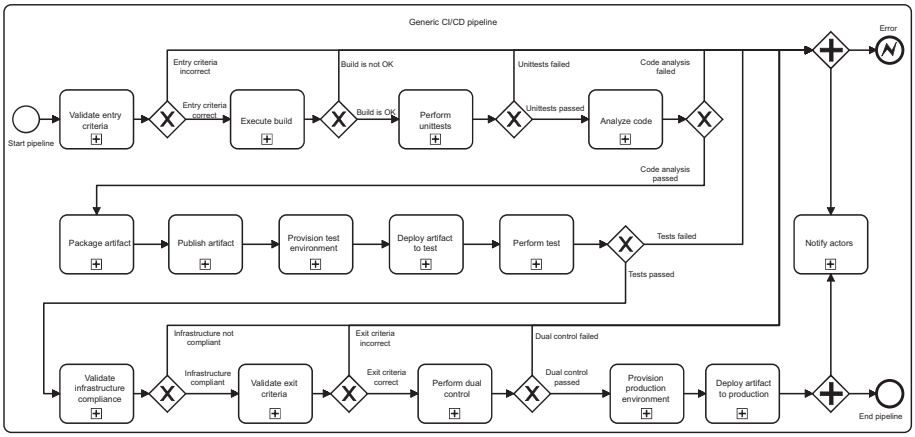


Figure 4-6. *BPMN, the Generic CI/CD Pipeline*

As you can see, Figure 4-6 shows the stages of the Generic CI/CD Pipeline, most of them ending with an exclusive gateway. The exclusive gateway is a condition that determines whether the stage result was successful. The pipeline either ends in a success state or ends an error/failed state.

The Generic CI/CD Pipeline consists of the following stages.

Validate Entry Criteria

A pipeline is triggered by a certain event that occurs in another system.[2] This often means that an API of the ALM platform/integration server is called from that system. For example, the trigger can be a scheduled event, a manual event (the pipeline is manually started), or any SCM event like Git push, merge, tag, etc. Any webhook implemented by an external

[2] Or it is triggered manually, of course.

system that calls the ALM platform/integration server API acts as a trigger.[3] Pipelines can call other pipelines, and it is even possible to hook up an advanced AI monitoring system to your production environment that detects deviating behavior in the application. This may result in triggering a pipeline to reconfigure the application or performing remediating activities on the infrastructure.

To make sure that the pipeline is started by a valid trigger using the correct trigger data and the correct pipeline configuration, a validation stage—the *Validate entry criteria* stage—is added to the Generic CI/CD Pipeline. The pipeline can proceed only if certain criteria are met. The following are typical entry criteria validated in this stage:

- Validate all mandatory pipeline variables in the *Validate entry criteria* stage. If one of the variables is not (properly) configured, the pipeline stops in the first stage instead of somewhere at the end of a pipeline run.
- Add a ping task to the *Validate entry criteria* stage to make sure that an external system is reachable. The ping task could send an HTTP request to an external system and validate the returned HTTP status. If, for example, a status 503 is returned, the pipeline stops, because the external system cannot be accessed.
- The branch—passed as an argument in the trigger—for which a release candidate is going to be built is indeed the expected branch. For example, only triggers with a Git event associated with the main branch are allowed, if the intention is to create a release.

[3] On an infrastructure level, this also means that the external system calling the API of the ALM platform must be an authenticated system. So, connections should make use of mTLS, OpenID Connect, or at least some basic authentication.

- Validate whether data passed by a pipeline trigger meets certain conditions in another external system. For example, validate whether the code has a reference to an existing work item in the issue tracker. On ALM platforms, these types of validations are often easy to configure. It becomes more complex if the integration platform consists of multiple independent tools.

Execute Build

This stage involves building artifacts from code, such as the creation of a `.jar` file from Java code or an `.exe` file from C++ code. The code associated with a certain branch and certain commit is checked out in the SCM system, dependencies are downloaded (for example, Java libraries from Maven Central), and the code is compiled. This is a fully automated process.

Perform Unit Tests

Unit tests are automated tests to make sure that components within a service or application behave as expected. Unit tests are usually isolated and independent. In principle, unit tests strive for 100 percent code coverage.

Note Although the Generic CI/CD Pipeline defines that, as a safeguard, the pipeline stops after a failure in the unit test, some people may decide to implement it differently and continue after a failure. This is to further validate the code and identify any issues that may not have been caught by the unit tests.

Analyze Code

The *Analyze code* stage provides confidence that the code quality requirements are met. Organizations often demand a combination of checks, sometimes completed with specific validations. Here are some examples:

- *Code quality assurance*: Static analyzers that assure the quality of the software, for example, SonarQube or SonarCloud to perform static analysis of code to detect bugs and code smells, OpenClover to validate code coverage, and Pylint to analyze Python code.
- *Static application security testing (SAST)*: Secure software by reviewing the source code of the software to identify sources of vulnerabilities. Tools are, for example, Checkmarx and Fortify Static Code Analyzer.
- *Software composition analysis (SCA)*: Automated scans of an application's codebase to identify security vulnerabilities and the type of license of all open-source components used in the build process. These types of scanners can detect whether an artifact contains a vulnerable version of log4j, for example. Tools like Nexus IQ or JFrog Xray fill in this segment.
- *Credentials scan*: This is an extension of SAST and scans other types of files for credentials, passwords, tokens, or other secrets, which are present in a code repository in plain text. Whispers is an example of such a tool. Whispers can detect hard-coded credentials in (property) files.

- *Validation of IaC*: This applies to the configuration of the infrastructure code, such as AWS CloudFormation or Azure ARM templates, and validates whether the configuration complies with certain company policies. Misconfigurations of the infrastructure are detected by analyzing the infrastructure code. One of the organization's policies could be that public access to an S3 bucket in AWS must always be blocked. If the infrastructure code defines that public access to a bucket is not blocked, it is detected by this pipeline task, which causes the pipeline to break.
- *Validation of pipeline code*: Even pipeline code must comply with certain quality criteria and policies. For example, the pipeline must contain certain SCA or SAST validations because they are mandatory by company policy. However, this type of validation is a bit odd because it does not validate the application code, but the pipeline code; the pipeline validates itself so to say. Integrating pipeline compliance validations in the pipeline itself is good, of course, because they immediately detect whether the pipeline is compliant, but to guarantee that pipelines comply with certain policies, the validations must be performed "outside" the pipeline, integrated into the ALM/integration platform.

The *Analyze code* stage may contain multiple tasks that potentially delay the pipeline, because some of these tasks can be very slow. Subsequent chapters point out what the options are to mitigate this.

Package Artifact

Packaging an artifact involves all activities to deliver an artifact that can be deployed to a test or production environment. Think of `.zip`, `.jar`, or `.exe` files. This also involves the creation of custom packages in cases where a dedicated deployment tool is used.

To guarantee the integrity of the artifact, specific measures must be taken, such as signing a package,[4] to make sure the artifact deployed to production is not compromised. For auditability, this is the point at which we want to ensure that the package goes to production unchanged.

Publish Artifact

Publishing an artifact means that the artifact is stored in an immutable binary repository such as Artifactory, Nexus, or Azure DevOps Artifacts. Docker images are pushed to a Docker repository, for example, Nexus 3 and AWS Amazon Elastic Container Registry (ECR).

Publishing an artifact is typically the last stage of continuous integration, and this is where continuous delivery begins.[5] The continuous delivery stages retrieve the artifact from the repository and use it for testing and deployment to production. This ensures that the same artifact is used throughout all environments and not built for every environment separately.

In addition to the published artifacts, additional information—metadata of the continuous integration process—can be published. The version of the artifact, the commit hash of the code, the work items that are part of the artifact, the developer of a feature, the pull request reviewer(s),

[4] Signing a package means that a digital signature is created and added to an artifact, to guarantee the integrity of the artifact.

[5] Continuous delivery is sometimes used as overarching concept that includes continuous integration.

and the unresolved but accepted issues are typical examples of metadata gathered during continuous integration. This type of information can be seen as the "contract" with the continuous delivery part of the pipeline, and it makes sense to gather this kind of metadata and publish it as a "release note" in a central place where all interested parties can read it. If needed, test results can be added later to this metadata, so it becomes clear whether a release candidate is suitable for production (or not). This metadata can also be used to determine whether the artifact has gone through all the mandatory steps before it is deployed to production.

Provision Test Environment

Infrastructure consists of several layers. The lowest layers may refer to installing physical hardware or requesting cloud accounts or subscriptions. These activities are not part of the *Provision test environment* stage. It can also be argued that shared infrastructure components, which are created once and almost never touched upon, should be moved to a separate base infrastructure pipeline. Base infrastructure involves, for example, DNS records, virtual networks, and subnets. The highest infrastructure layer typically contains infrastructure components, associated with a business feature (and the application). Think of queues, a file system, and a database.

The infrastructure components are created on the fly using infrastructure as code (IaC). This results in the creation of a test environment, which can be destroyed again after the tests are executed; this is called *ephemeral infrastructure*. The execution of the IaC code should be idempotent, meaning that if the same code is executed twice and not changed in between, nothing changes in the target environment.

An ephemeral test environment has the benefit that it reduces costs because you pay only for what is used, and the tests always have the same starting position; they start with a new and clean test environment. However, be aware that it is not always beneficial to create and delete a test

environment on the fly. In cases where you can make use of infrastructure as code—for example, in the cloud—it is relatively easy to create an infrastructure, but there still may be some issues. Consider, for example, a long creation time of your infrastructure, or deleting your stacks is problematic because they have dependencies with resources that cannot be deleted or at least not easily. Also, test environments used for load and performance tests cannot be deleted so easily because they often contain very large databases. Rebuilding the environment would take several hours. And if a test environment is almost continuously used, it makes no sense to tear down the environment and rebuild it a second later. That is why organizations still make use of fixed test environments, even if they are created using IaC.

On the other hand, keeping test environments intact and leaving them unused for a longer time should be avoided. Teams decide whether to create a test environment once and use it for a longer time and destroy it if not needed anymore or not needed shortly. Also, a combination of more or less permanent test environments combined with ephemeral test environments is possible.

Note If only Docker containers are used, the situation is a little different. The Docker containers represent the test environment, and they can easily be created and removed. However, the whole Docker runtime environment—the base infrastructure—itself (for example, Kubernetes) remains.

Deploy Artifact to Test

Deployment involves all activities required to install the software on a target environment, so it can be tested. This may involve deployment to one or more test environments. It also depends on the type of testing

to be performed. System tests, for example, need a smaller-sized test environment compared to test environments in which load, stress, and performance tests are executed. The *Deploy artifact to test* stage includes all deployments of the artifact to all required test environments.

Perform Test

Testing covers a wide range of types from contract tests and integration tests to usability tests and production acceptance (preproduction) tests, except for unit tests, which are performed in a dedicated stage. More details concerning the different test types are discussed later in this chapter.

It is important to point out is that tests should not rely on each other. Each test must be able to be performed individually, which offers the possibility to perform tests both sequentially and in parallel. Each time a test is executed, it is initialized to a certain starting point.

Validate Infrastructure Compliance

Validate infrastructure compliance is a bit of an odd stage. It is an addition to the *Validation of IaC* task in the *Analyze code* stage. The *Validation of IaC* task is a static code analysis task of the infrastructure code. The *Validate infrastructure compliance* stage involves a dynamic scan of the target environment, the application running in this environment, resources used by the application, and application-specific settings. The scanning is performed according to security compliance rules (guardrails). The stage checks whether certain (unused) ports are open, whether restricted protocols are used (e.g., HTTPS in favor of HTTP), and whether protocols are configured but not used by the application. The list of checks can be very long.

The reason why this is a separate stage, executed only after all tests have been performed, is that the focus of the pipeline flow should be first on testing whether the application works properly and second on whether the infrastructure resources associated with the application are compliant.

Note There is also an overlap between static scanning of the infrastructure code (as part of the *Analyze code* stage) and dynamic scanning of the infrastructure (as part of the *Validate infrastructure compliance* stage). Both could check the same configuration, but a dynamic scan proves that a certain configuration is also reflected in the target environment in the way it was meant. A recommendation is to perform both, if possible.

Validate Exit Criteria

Validating exit criteria can be considered as a gate that determines whether the artifact is allowed to be deployed and installed in the production environment. Some of these validations determine whether the artifact was built as expected, and other validations are mandatory because the production target environment itself also assesses whether the deployed artifact meets certain criteria before it is installed. Here are some examples of exit criteria validations:

- It is not allowed to install software without a valid digital signature (nonsigned software) because it must be guaranteed that software can be deployed and installed from an authenticated pipeline. This prevents someone from trying to install software manually. If this criterion is not met, the target environment does not allow installation or prevents the software from

execution. This kind of feature is present in Windows Defender Application Control (WDAC) on Windows, for example. The *Validate exit criteria* stage makes sure that it does not come to this and stops the pipeline if the software is not digitally signed.

- Another example is that an artifact must be tagged or versioned; otherwise, it is unknown which version has been deployed, and the artifact cannot be identified in production anymore.
- An artifact may be built only from the main branch or a release branch. If the artifact somehow turns out to be built from another branch, deployment is not allowed.
- The artifact is a release candidate that originated from the main or release branch, but according to the metadata associated with the artifact, it did not pass all tests.
- The version of the artifact to be deployed is higher than the version of the artifact running in production.[6]
- There is a *change freeze*. It is not allowed to deploy to production during the change freeze period. If the pipeline detects that the deployment is started in the change freeze period—which is configured in the ALM platform/integration server—it aborts the deployment.

[6] This applies only to regular deployments and is not a rollback to a previous version because of an incident.

- The artifact is expired. Even if the version of the artifact looks fine, the artifact is built from the main or release branch, and all tests are passed, deployment still may be aborted because the artifact is expired. If an artifact is "too old," it may pose a risk if deployed to a production environment.
- The artifacts contain development tools, test code, or testware. They could potentially cause issues or expose sensitive information.

In principle, the exit criteria of the pipeline are the entry criteria of the target platform. It makes sense to validate the artifact to determine whether it does comply with the preconditions of a deployment (to production), especially in cases in which more teams build artifacts for a shared production environment. The positioning of this stage before the actual deployment to production also makes sense because that is the last possible moment to validate the artifact before it is deployed.

Perform Dual Control

An artifact may be deployed only if it was approved by a release manager, a product owner, or a designated person (a delegate). This approval is called *dual control* because there is always a second person involved in putting an artifact into production. This approval is a manual task.

Performing a dual control is conceptually part of the *Validate exit criteria* stage, but it is modeled as an explicit stage in the Generic CI/CD Pipeline. It is such an important step in the process, it is made explicit.

This is by definition also the only manual step in the process; all other stages and tasks are automated.[7] Having the dual control stage in the Generic CI/CD Pipeline also makes sense; otherwise, the pipeline would have become a continuous deployment pipeline and not a continuous delivery pipeline.

Provision Production Environment

This is the same stage as the *Provision test environment* stage but now for the production environment. This also means that the same IaC is used for both the *Provision test environment* stage and the *Provision production environment* stage. The only differences are the target environment and the environment-specific properties and resources (e.g., certificates).

Deploy Artifact to Production

This stage is performed if nothing stands in the way anymore to deploy the artifact to production. The artifact is retrieved from the binary repository and installed in a production environment. This includes any configuration change needed in the production environment itself. Technically, there are various solutions to deploy software, from executing an `scp` (Linux) command to securely transferring and installing files to a production server using a dedicated deployment tool.

The implementation of this stage also depends on the deployment strategy. A re-create deployment strategy results in a different design and implementation than a blue/green deployment strategy.

[7] In theory, of course. Often there are still manual test tasks to be performed.

Notify Actors

This stage has a generic name. It deals with informing team members about the pipeline execution result, both success and failure, but it also deals with notifying other actor types about the result. Other actors are, for example, external systems, other pipelines, or specific functions of the ALM platform/integration server. Informing actors can be implemented as simply as sending an email to the team or a more sophisticated activity such as performing an outbound API call to an external system.

Note The Generic CI/CD Pipeline model suggests that the *Notify actors* stage is called only if all pipeline stages are executed and the pipeline ends. That is not true. Each stage has the responsibility to perform fast feedback and notify its actors. In the model, this is delegated to the *Notify actors* stage.

Design Strategies

As the previous chapter already shows, there are lots of possible requirements and aspects that influence the design and realization of a pipeline: the business organizations' software delivery strategy, the workflow of the team, security aspects, and certain constraints, both technical and nontechnical, etc. In the end, the pipeline design and realization are derivative products of all these aspects, and if one of them is suboptimal, the pipeline is also suboptimal.

It is important to have a continuous interaction between optimizing the requirements on one hand and the design and realization of the pipelines on the other hand. If, for example, the team's workflow is overly

complex, it puts a burden on the software supply chain. An optimized workflow should lead to a smooth and fast software delivery process resulting in an optimized pipeline design. See Figure 4-7.

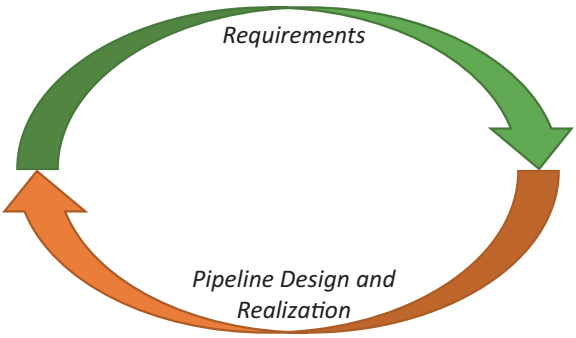


Figure 4-7. *Pipeline design and realization cycle*

The next couple of paragraphs handle some common design strategies, which deal with specific requirements or constraints and visualize how they shape the design of the pipeline. Although these paragraphs form only a subset of all possible cases, they still provide a nice profile of various situations.

Context Diagram

Although the design phase is abstract, it does make sense to draw a context diagram containing all actors. Actors are not only the people who are involved but also the surrounding systems. A context diagram gives an impression of which interactions take place in the context of CI/CD. Include everything you already know—including tools—in the context diagram and use abstract names like SCM, issue tracker, and the SCA tool, if you do not know which tools are used (yet). A context diagram might look something like Figure 4-8.

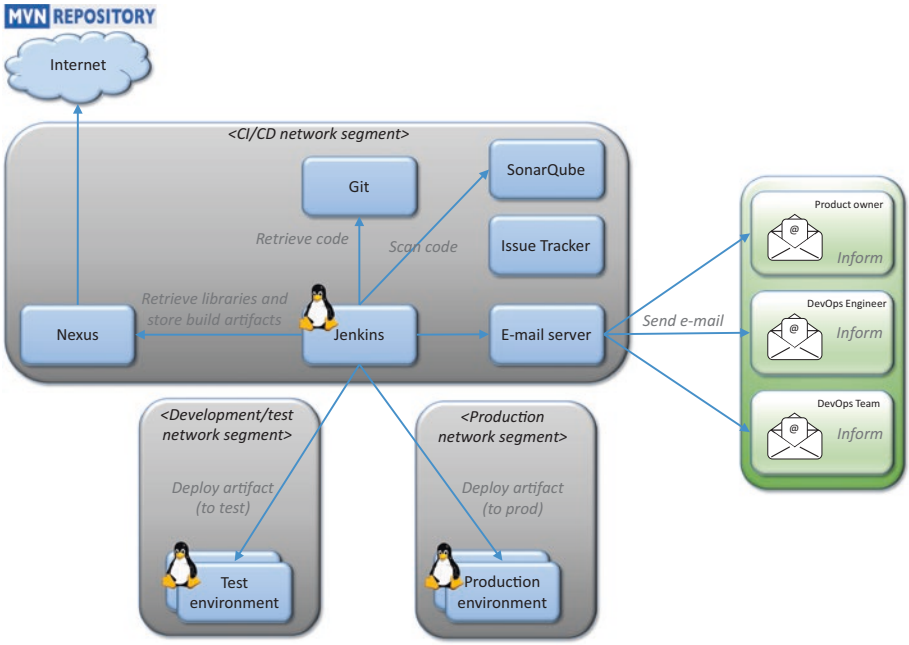


Figure 4-8. *Context diagram*

Figure 4-8 shows a central Jenkins setup running on a Linux server in a special CI/CD network segment. The target environments are also Linux clusters: one production cluster in the production network segment, and a cluster in the development and test network segment. In this diagram, the Jenkins server retrieves code from Git and libraries from Nexus, which is connected to an external Internet source, Maven Central. Application code is scanned with SonarQube, and artifacts are stored in the Nexus repository. Information about the pipeline status is shared—using email—with three types of roles, the product owner, the product owner delegate, and the DevOps engineers. The context diagram also shows an issue tracker, but this is a stand-alone system and not connected to any of the other systems. It is therefore not possible to automatically check whether a work item is present in the issue tracker.

A context diagram is a good way to discuss with the team how the pipelines interact with all actors. The first version of the context diagram is probably a simple diagram with some blocks like the previous one, but the diagram is extended along the way, with more (technical) details added in later versions. Use the context diagram in the discussions with the team to point out what is added or changed in the pipeline setup.

Branching Strategy

A branching strategy is a critical element in the way a pipeline is shaped. At the start of a pipeline design, it must be clear how the team works and which workflow they adopt. Depending on the type of strategy, the pipeline flow differs. Some of these strategies are discussed in the next paragraphs and demonstrate what a possible pipeline design could look like.

Trunk-Based Workflow

In the context of continuous integration, there is only one workflow, the trunk-based workflow. All other strategies are not considered continuous integration, but they are relevant because lots of teams still use a branch-based workflow.

The trunk-based workflow model is the simplest workflow strategy. This means that the source code repository (e.g., Git) contains only the main branch, the trunk. Changes are directly applied to the trunk, and also release candidates are created from the trunk. The complexity of a trunk-based pipeline is relatively low compared to other branching strategies. See Figure 4-9.

Figure 4-9. *Trunk-based workflow*

A developer works on a local copy of the trunk and commits its changes (locally). As soon as the developer has completed their work, the code is pushed to the remote trunk. That is the moment the pipeline starts running, with the intent to deploy the finished work to a production environment. This means that in the case of a trunk-based workflow, the main branch is always in a production-ready state. A pipeline associated with a trunk-based workflow covers all the stages of the Generic CI/CD Pipeline. See Figure 4-10.

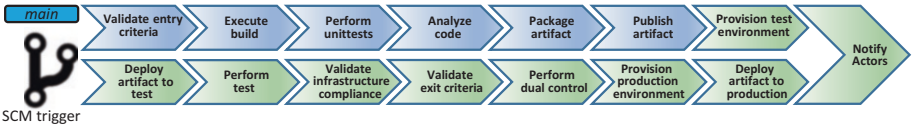


Figure 4-10. *Trunk-based pipeline*

What does the workflow look like in practice? Most likely, some kind of issue-tracking software like Jira is used to register work items. A work item—also called a *user story* or a *project backlog item*—defines the feature that needs to be built. This feature must be small, preventing the merges of large pieces of code. Keeping the trunk "clean" requires disciplined commit hygiene, and big changes to the trunk must be avoided.

The trunk-based workflow fits perfectly in a pair programming way of working. In pair programming, two developers are working on a local copy of the trunk and pushing their software code directly to the trunk. This results in a release build that can be deployed to production if all intermediate stages are passed.

This workflow makes pull requests obsolete because there isn't a separate branch and reviewing the code is done on the spot. This also reveals an issue with the trunk-based workflow. If not done properly, code reviews are not administered, and it becomes difficult to trace back the input of colleague developers.

The traceability of a change is important. The work item and code commit are related, and it must be clear which work item has led to a certain code change. Most integrated ALM platforms include features that take care of this. If individual CI and CD applications are used, it becomes more difficult to establish this relation.

Use case:

A team uses a trunk-based workflow and uses Git as their SCM system. Team members perform pair programming, which involves two developers per development session. The review is done by both developers during development, and one of them performs the commit/push. There is an organizational audit requirement that states that all users who reviewed the code need to be registered. It must be possible to trace back the code commit to a work item. The team uses an issue tracker system. In this particular case, the test and production environments are already provisioned.

Considering this use case, a design of a trunk-based workflow contains the following ingredients:

- The commit must contain a well-formatted comment with a work item reference and the involved developers. This can be solved by adding a comment to a code commit and enforcing that the comment is well-formatted. This policy enforcement can be established using a server-side Git hook, which forces code comments such as the following:

  ```
  git commit -am "- feat(JIRA123): fixed nullpointer
  exception - authors: John, Frank"
  ```

An alternative is the use of a review system such as Gerrit [17] where a push is intercepted and other people can review changes before they are applied to the trunk.

- The work item must exist in the issue tracker.
- The trigger must contain the branch for which the commit and push are performed. This branch must be the main branch (trunk).

Modeling this case in BPMN notation results in a design like in Figure 4-11.

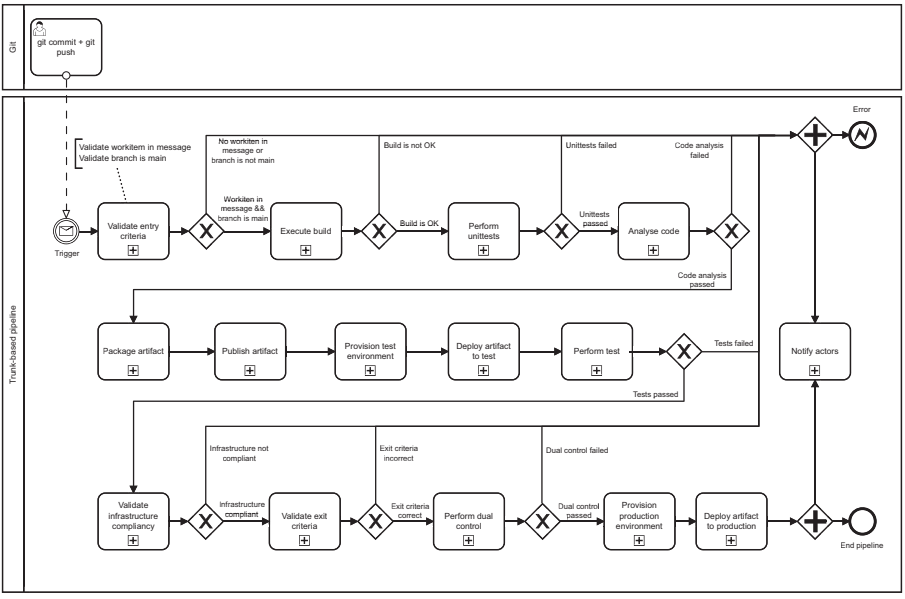


Figure 4-11. *BPMN, trunk-based pipeline*

This diagram resembles the Generic CI/CD Pipeline, with some minor additions. Added to the diagram is a specification of the stage *Validate entry criteria*. The first task in this stage is to determine whether the branch to which the code was pushed is indeed the main branch. The stage

also contains tasks to validate whether a commit contains a work item reference and who were the developers reviewing the code. A check to determine whether the work item exists is also included.

Zooming in on the *Validate entry criteria* stage results in the detailed model shown in Figure 4-12.

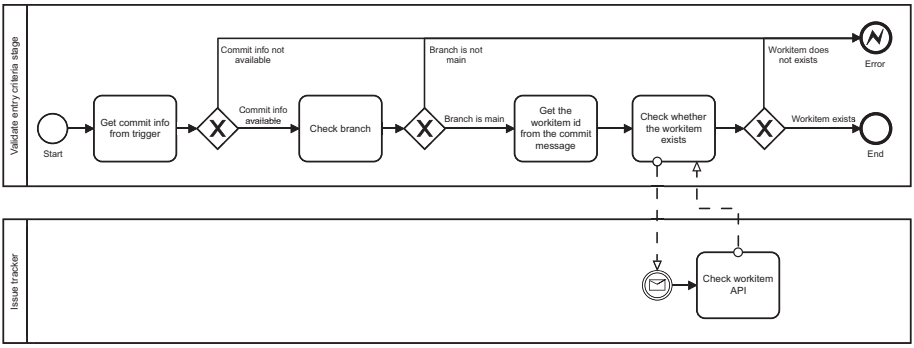


Figure 4-12. *BPMN, validate entry criteria tasks*

The *Validate entry criteria* model defines several tasks.

- *Get commit info from the trigger*: The trigger consists of an API call (webhook) to the ALM/integration platform. The pipeline was triggered after a code push in Git. The commit info is passed in the request of the API. In this case, the commit message and the branch name are expected.
- *Check branch*: The branch name is included in the commit info and passed to the pipeline through the trigger. The branch name is validated and must be main.
- *Get the work item ID from the commit message*: The code commit message is parsed to get the work item ID (Jira123) from the message.

- *Check whether the work item exists*: If the ALM platform/integration server and the issue tracker are not integrated into one system, this task involves an API call to a remote issue tracking system to validate whether the work item exists. The result of this API is used for the subsequent flow of this stage.
- *Check work item API in issue tracker*: The issue tracker API queries whether a certain work item ID exists.

Modeling the pipeline stages and tasks isn't that complicated, but explicitly designing it makes you more aware of the whole process, the tasks involved, and what exactly needs to be implemented. Because the trunk-based workflow results in a more or less straightforward pipeline model, it is the preferred workflow of many teams. There are some alternatives to the trunk-based workflow, like a trunk with a separate release branch, but the principle of the workflow remains the same; you directly push your commit to the trunk.

As shown in the next paragraphs, the pipeline design becomes more complex as the complexity of the workflow increases.

Feature Branch Workflow

Despite Dave Farley's statement that you shouldn't use branching [28], it is still used a lot. Feature branch workflow is one of the alternatives to a trunk-based workflow. This means that the repository consists of the main branch—the trunk—and from the main branch separate feature branches are spawned. The main branch is a permanent branch, while the feature branches are short-lived branches in which a business feature is developed. Feature-based branching models are not considered continuous delivery unless the features are really small. See Figure 4-13.

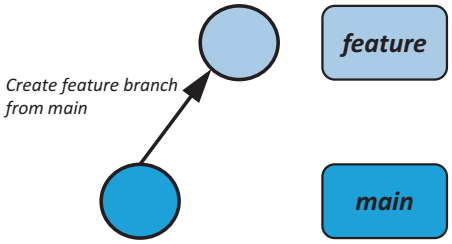


Figure 4-13. *Feature branch workflow*

The developer commits code to the feature branch. This can be done several times. If the feature is completed, they create a pull request, so other developers can review the code. If the colleague developers approve the pull request, the code of the feature branch is merged back to the main branch.[8] See Figure 4-14.

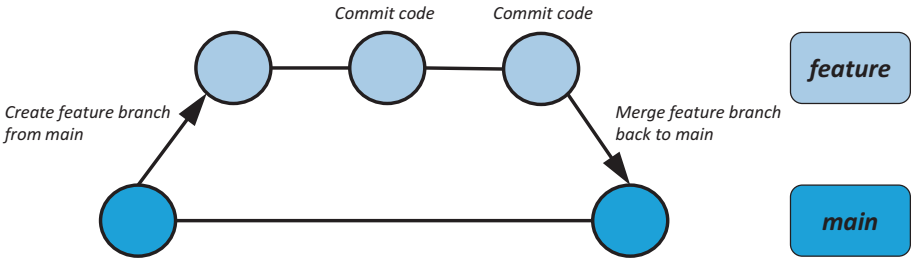


Figure 4-14. *Feature branch workflow, merging the feature branch into the main branch*

A design principle that works out very well is that "Each branch has an associated pipeline." The reason is that each branch has its purpose and its life cycle, so why would the pipeline execution be the same for different types of branches?

[8] From a technical (Git) point of view, you can decide to merge the *feature* branch back to *main*, or rebase *main* onto the *feature* branch, to get a cleaner history. In addition—if the platform supports it—you may define branch policies on the main branch to prevent, for example, that a feature is merged that does not even build successfully.

In a feature branch workflow model we deal with two types of branches. A developer working on a *feature* branch will commit/push a couple of times during the day and merge back to the *main* only at the end of the day.

If a push to a remote feature branch is done often, feedback from the pipeline toward the developer is expected to be fast. It does not make sense to execute the whole cycle of build, quality assurance, deployment, and test each time a developer pushes code to a feature branch. And if you also add the provisioning of an ephemeral infrastructure into the equation, this whole cycle just takes too long.

A practical approach to this is to limit the number of stages of a pipeline triggered by an activity on a feature branch. Often a few stages are sufficient to demonstrate that the artifact can be built and unit tests are performed successfully. See Figure 4-15.

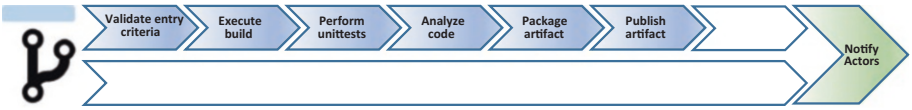


Figure 4-15. *Feature branch workflow, feature branch pipeline*

The pipeline associated with this feature branch looks like Figure 4-16 in BPMN notation.

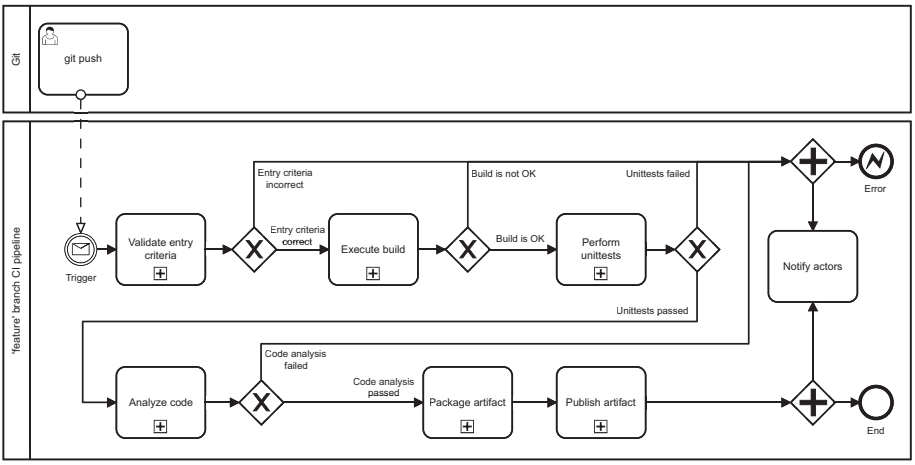


Figure 4-16. *BPMN, feature branch workflow; feature branch pipeline*

The pipeline associated with a feature branch is a CI pipeline and not a full CI/CD pipeline. Provisioning of infrastructure and testing—except for unit testing—is not part of this pipeline, which makes it lean and mean and limits the use of resources of the ALM/integration platform. If the CI pipeline associated with this feature branch executes successfully, the developer is allowed to create a pull request. If the CI pipeline does not execute successfully or the quality of the code is not sufficient, it does not make sense to create a pull request because colleagues will not approve code that does not build.

Creating a pull request allows co-workers the opportunity to review the code; if they approve, the feature branch is merged back into the main branch, and the pipeline of the main branch starts. This pipeline traverses through all the stages of the Generic CI/CD Pipeline. Similar to the trunk-based workflow, the main branch in a feature branch workflow must be production-ready. It is the main branch from where a release is created. See Figure 4-17.

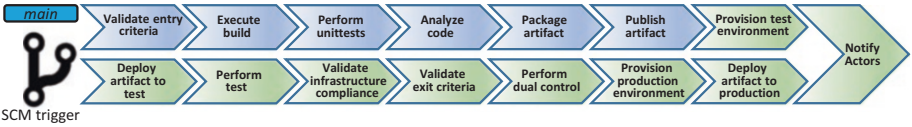


Figure 4-17. *Feature branch workflow, main branch pipeline*

The pipeline of the main branch, modeled in BPMN notation, looks like Figure 4-18.

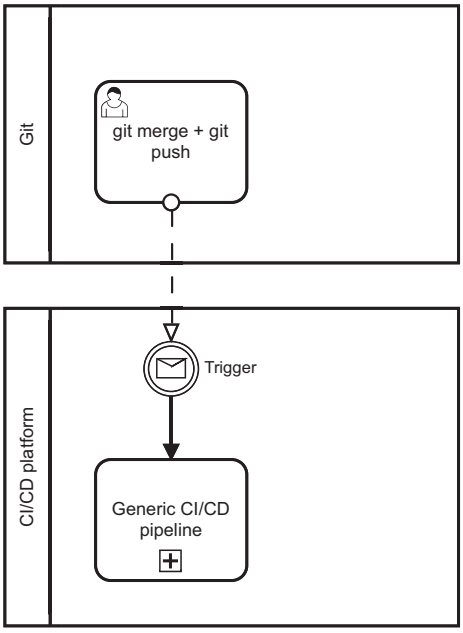


Figure 4-18. *BPMN, feature branch workflow; main branch pipeline*

Table 4-2 summarizes the tasks performed for, respectively, the feature and main branch. This is just a proposal, and of course, it is perfectly fine to deviate from it. Essential, however, is to think about the stages that are executed for each branch and why.

Table 4-2. *Feature Branch Workflow—Branches vs. Pipeline Stages*

Branch	Stages to execute	Rationale
Feature	• *Validate entry criteria* • *Execute build* • *Perform unit tests* • *Analyze code* • *Package artifact* • *Publish artifact* • *Notify actors*	The reason to execute only the CI stages is that the response to the developer must be almost immediately. It happens often that a build succeeds on the developers' local machine, but not in the pipeline. The feature branch pipeline is the first step to making sure that the code can be built in a pipeline. In addition, the code of a *feature* branch is committed frequently (to the remote server). To minimize resource consumption, only the proposed stages are executed.
Main	• *Validate entry criteria* • *Execute build* • *Perform unit tests* • *Analyze code* • *Package artifact* • *Publish artifact* • *Provision test environment* • *Deploy artifact to test* • *Perform test* • *Validate infrastructure compliance* • *Validate exit criteria* • *Perform dual control* • *Provision production environment*	The pipeline associated with the main branch creates a release (candidate) artifact. This artifact is tagged and versioned as a release artifact. All stages of the *Generic CI/CD Pipeline* are incorporated into the pipeline.

Note on Implementation

This book intends to be abstract and tool-agnostic as much as possible. In cases where implementation is discussed, the technical details are kept to a minimum. But there are some pointers concerning the realization of the pipelines.

On a design level, two pipelines are distinguished, one associated with the feature branch and one associated with the main branch. Of course, it is perfectly possible to develop two pipeline implementations, but it is also possible to realize one technical pipeline integrating both logical pipelines. The technical pipeline makes use of a condition to distinguish between branches and uses templates or libraries to reuse stages. In Azure DevOps, for example, validating a branch can be defined as follows:

```
condition: eq(variables['Build.SourceBranch'], 'refs/
heads/main')
```

This condition determines whether a specific section in the pipeline is executed only if the current branch is main.

A combined pipeline of a feature and main branch in BPMN notation looks like Figure 4-19.

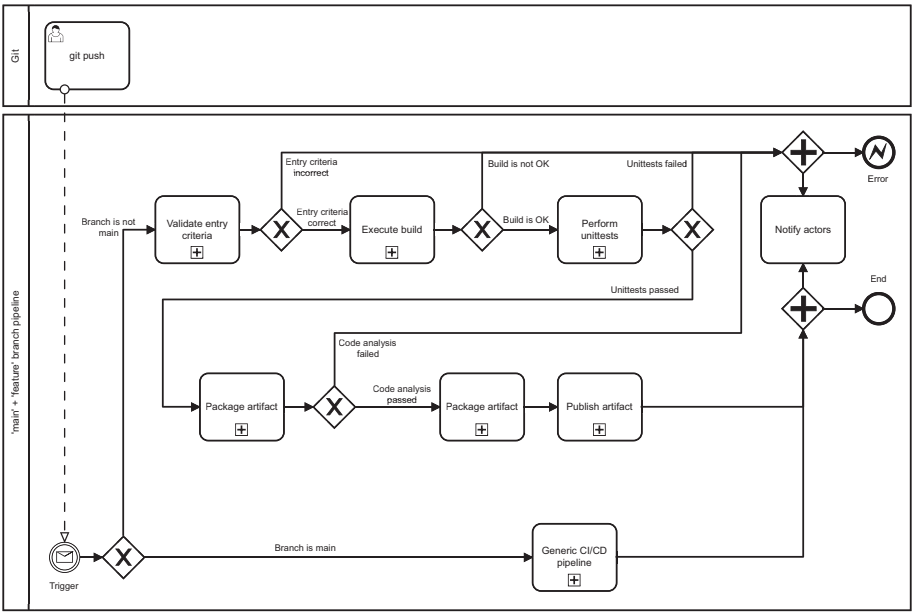


Figure 4-19. *BPMN, Feature branch workflow; pipelines main and feature combined*

The model starts with a condition to determine whether the Git push came from the main branch or not. If true, all stages of the Generic CI/D pipeline are executed. If false, only a subset of these stages is executed.

Gitflow

Gitflow is still used by a lot of teams. It was one of the first workflows developed and is still popular.

The repository consists in its core of two branches, master and develop. These branches have an infinite lifetime. The master contains all code that is deployed to production. The deploy branch contains the code that reflects the current state the team is working on. In recent workflows, the name *main* branch is used in favor of master. To keep aligned with the previous paragraphs, the name *main* branch is used in the remaining chapters of this book.

If a developer starts to work on a feature, a feature branch is created from the develop branch. The usage of the feature branch is similar to how it is used in the feature branch workflow; instead, it originates from the develop branch and not from the main branch. See Figure 4-20.

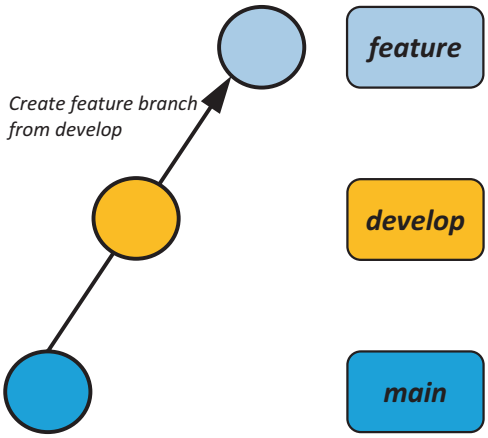


Figure 4-20. *Gitflow, create feature branch*

As soon as a feature is completed, the feature branch is merged back into the develop branch. This happens multiple times, so the develop branch is always [0..x] features ahead of the main branch.

As soon as the code in the develop branch reaches a stable situation, all tests are performed successfully, and the team is convinced that the code is in a production state, so a release candidate is created. Instead of directly merging the code from develop to main, an intermediate branch is created: the release branch. The release branch contains all the code of the release candidate, ready to be deployed to production. The release branch is temporary and is used to align with the release version. It is not a finalized version yet. It is still possible that the code of a release branch is updated, but this should include only small bug fixes.

To keep the main branch always in a state that reflects production, the release branch is merged back into the main branch after the code is deployed to production. This is also done for the develop branch. See Figure 4-21.

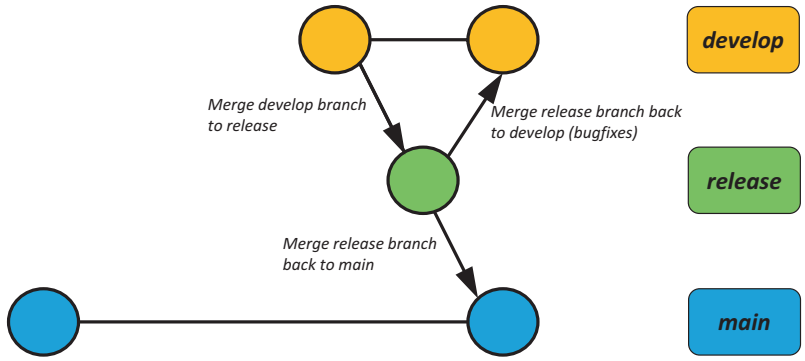


Figure 4-21. *Gitflow, the release branch*

Hotfix branches are used to fix a bug in production. Instead of using the regular workflow that includes feature, develop, and release branches, the hotfix branch is based on the main branch and is not derived from the develop branch. After the hotfix is tested, approved, and deployed to production, it is merged both into the main and develop branches. See Figure 4-22.

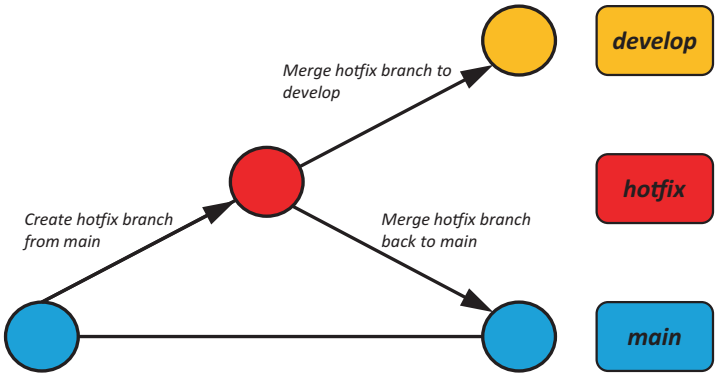


Figure 4-22. *Gitflow, the hotfix*

Summarized, Gitflow involves five different branch types (see Figure 4-23).

Main (or master): This branch always contains the actual production code.

Develop: This branch includes all code of the main but is normally ahead of the main branch. It includes features of the upcoming release.

Feature: Feature branches are short-lived branches, containing the code of each feature. The code is merged with the develop branch after a pull request has been opened and approved by other developers.

Release: Release branches are based on the develop branch and are created as soon as a release candidate must be created. After the release branch is created and finalized, it is merged back into the main *and* develop branches.

Hotfix: A hotfix branch is created from the main branch and used to fix bugs in production. It is merged back into both the main and develop branches after it is successfully tested.

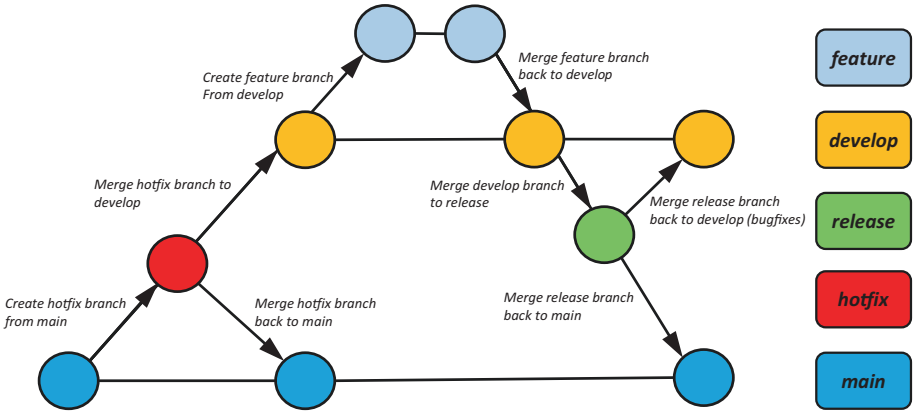


Figure 4-23. *Gitflow*

Because Gitflow works with five types of branches, it potentially results in five logical pipelines. Table 4-3 presents an overview—a proposal—of the branches and the associated pipeline stages, which are executed as soon as a pipeline is triggered.

Table 4-3. *Gitflow—Branches vs. Pipeline Stages*

Branch	Stages to Execute	Rationale
Feature	• *Validate entry criteria* • *Execute build* • *Perform unit tests* • *Package artifact* • *Publish artifact* • *Notify actors*	The reasons to create a pipeline with these particular stages are the same as the feature branch workflow, except for the lack of the *Analyze code* stage. This is omitted to provide even faster feedback and because it is present in the develop pipeline anyway.

(*continued*)

Table 4-3. (*continued*)

Branch	Stages to Execute	Rationale
Develop	• *Validate entry criteria* • *Execute build* • *Perform unit tests* • *Analyze code* • *Package artifact* • *Publish artifact* • *Provision test environment* • *Deploy artifact to test* • *Perform test* • *Notify actors*	Changes in the develop branch must be built and thoroughly tested because it potentially contains multiple features, all merged in the same develop branch. You may consider testing the functional aspects as part of the *develop* pipeline only, while all test types—including nonfunctional tests—are performed as part of the release pipeline.
Release	All stages of the Generic CI/CD Pipeline	A *release* branch is typically used to create a release artifact that is deployed to production. This justifies a *release* pipeline, containing all stages of the *Generic CI/CD Pipeline*.
Main	No pipeline	The sole purpose of the *main* branch is just to maintain the state of the code of the production situation. Unlike the other branching strategies, an artifact in Gitflow is not directly built and deployed from the main branch.
Hotfix	All stages of the Generic CI/CD Pipeline	A release in Gitflow is originated either from a release branch or from a hotfix branch. This justifies a hotfix pipeline, containing all stages of the *Generic CI/CD Pipeline*.

Figure 4-24 through Figure 4-27 are the pipelines associated with the branches of Gitflow.

Figure 4-24. *Gitflow, feature branch pipeline*

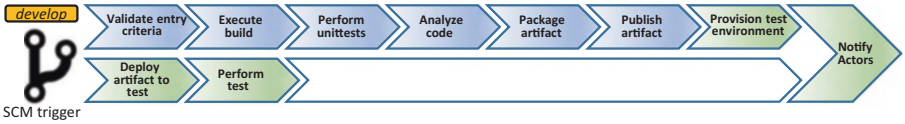


Figure 4-25. *Gitflow, develop branch pipeline*

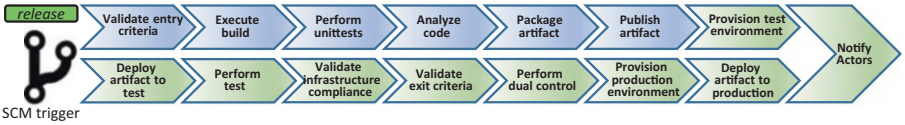


Figure 4-26. *Gitflow, release branch pipeline*

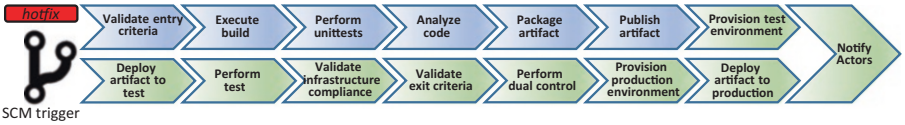


Figure 4-27. *Gitflow, hotfix branch pipeline*

All Gitflow pipelines are combined into one BPMN model, as shown in Figure 4-28.

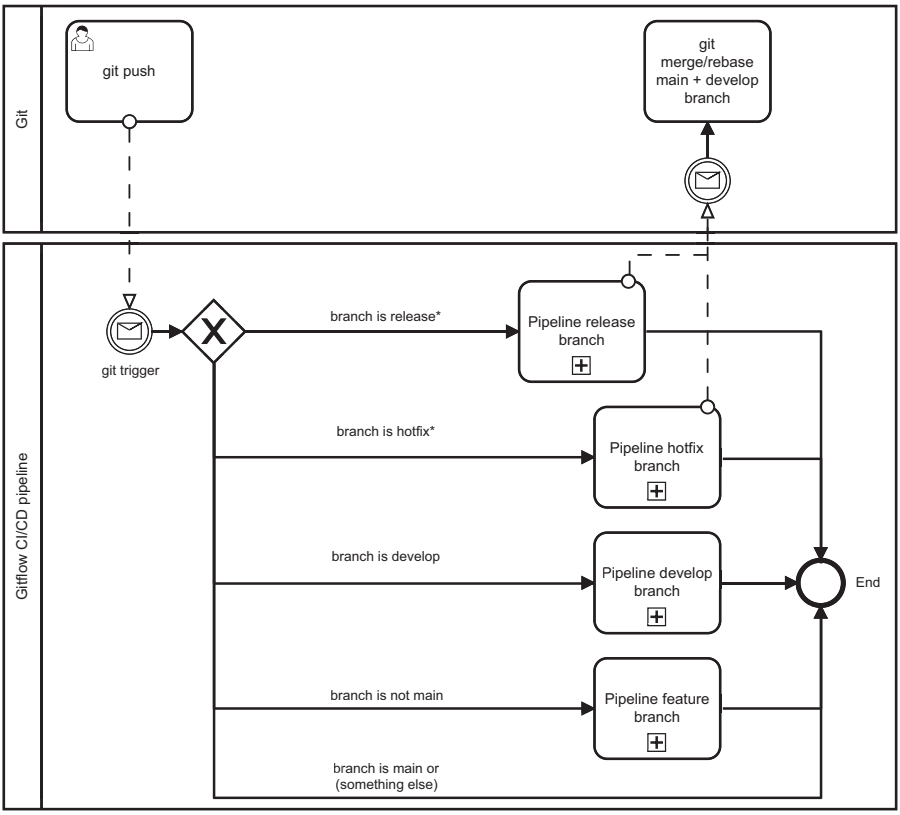


Figure 4-28. *BPMN, Gitflow combined pipeline*

As you noticed, the more branches there are, the more complex the workflow, which translates to the complexity of the pipeline design. Also notice that the "continuous" aspect becomes less if more branches are involved. The Gitflow model is not considered a proper model for CI/CD, because of its complexity, multiple—long-lived—branches, and slow-to-adapt new features because of a strict release cycle.

Build Strategy

One could argue there is not much to tell about building an artifact. You selected the appropriate build tool and apply the principle "Build once, run anywhere." In essence, your *Execute build* stage itself consists of just one task, often executing one line of code, for example:

```
mvn clean package
```

Or

```
msbuild mySolution.sln /t:Clean,Build
```

After a couple of minutes, the artifact is created, and that's it. But, in reality, the creation of an artifact has lots of aspects to be taken into account. Maybe the build lasts for 10 minutes, half an hour, or even longer. This breaks the "fast feedback" principle of CI/CD and asks for a strategy to decrease the build time. And other factors influence the build strategy or even shape the whole pipeline. What is the build strategy in case the target environment is the cloud, or what is the build strategy if there are multiple DevOps teams involved in the development of one integrated system? Let's highlight some of the factors associated with a build strategy.

Vertical Scaling

If the build time increases, vertical scaling is an option to speed up build times. Adding a larger server with a faster processor, more processor cores, and a faster disk is an option. But vertical scaling does not always help in the long run if more demanding builds occur. Other build strategies are needed, from which lots of advantages can be gained and which do not require any additional hardware.

Full Builds vs. Incremental Builds

One thing to take into account concerning build time is the execution of a full build versus an incremental build. In most pipelines, the build strategy that is chosen often is a full build strategy. This means that all application code is compiled and a completely new artifact is created every time the *Execute build* stage runs. If the build time is acceptable, this is a safe approach. If a full build takes too much time, an incremental build can be considered as an alternative. Figure 4-29 illustrates a full build.

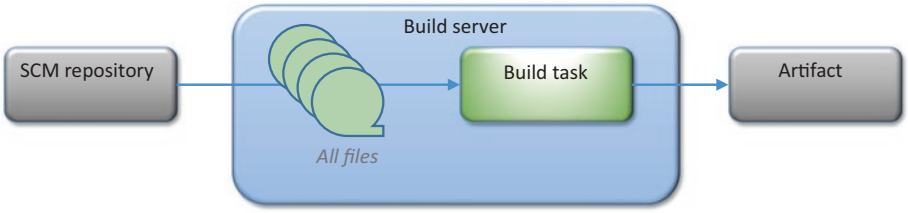


Figure 4-29. *Full build*

The concept of an incremental build needs a bit more explanation. Take a Java `.jar` file, for example. The `.jar` file is a build artifact composed of multiple `.class` files. Each `.class` file contains compiled Java code (bytecode). The source is plain Java code, stored in a `.java` file.

Another example is the compilation of C++ code. The C++ code is embedded in a `.cpp` (or `.h`/.hpp) file, resulting in an `.obj` (object) file after compilation. The final executable file consists of `.obj` files, all linked together to an `.exe` artifact (in the case of Microsoft Windows).

The trick with incremental builds is that only changed source code files are recompiled. If an executable is constructed from 500 `.obj` files, a full build recompiles all 500 `.cpp` files again, even if just one `.cpp` file is changed. An incremental build makes use of the output from

previous builds, and this results in the compilation of just the changed `.cpp` file. This speeds up the build time considerably. An incremental build is depicted by Figure 4-30. But there are a couple of caveats with incremental builds.

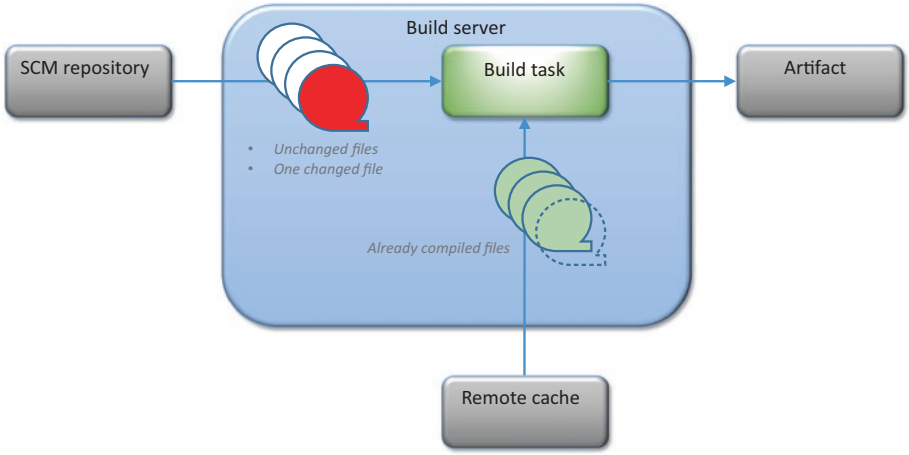


Figure 4-30. *Incremental build*

Incremental builds sometimes give some unpredictable results. It happens that an incremental build is screwed up somehow, and a changed file is not recompiled. This results in an unreliable artifact. Unfortunately, some build tools are a bit buggy, and these issues happen. This results in the fact that full builds are often preferred over incremental builds; better be safe than sorry. But this can be avoided. There are solid build tools that deal with incremental builds in the intended way.

Not all build systems support incremental build. Maven is a build system that does support some form of incremental build, and it was never created with the intent to perform incremental builds. So, when choosing an incremental build strategy, consider the appropriate build tool that was designed with incremental builds in mind. Gradle or Bazel are alternatives that natively support incremental builds.

Note From an audit point of view, one could raise concerns about which pipeline run is responsible for the creation of an artifact if incremental builds are used. The pipeline that creates the artifact running in production must be traceable. But if more pipelines are involved in the creation of the artifact, they all need to be part of the audit chain. Maybe the last pipeline run was responsible for only 1 percent of newly compiled code, while the other 99 percent was compiled by other pipeline runs, and although the chance that a clean—full—rebuild would deliver a different artifact compared to an incremental build is small, it is theoretically not null. In addition, it may even be difficult or impossible to point out which pipeline builds contributed to the creation of an artifact. To circumvent this issue and avoid difficult discussions with the Audit department, it may be wise still to use full builds if the build time is acceptable.

Parallel Builds

In addition to full and incremental builds, there is also the option of parallel builds. A parallel build spreads the compilation of source files over multiple threads (on one server) or even over multiple servers, depending on the platform setup. This results in the following strategies:

- *Multithreaded builds*: A multithreaded build makes use of the fact that a build tool uses multiple threads on one server to build an artifact. Build tools often include a flag that can be set to enable multithreading, even with the option to provide the number of threads or cores. A build can profit enormously from this feature; if multithreading is enabled and four threads are specified, it can make full use of a multicore CPU

architecture and compile multiple files in parallel. Multithreaded builds can be used in combination with both full builds and incremental builds. See Figure 4-31.

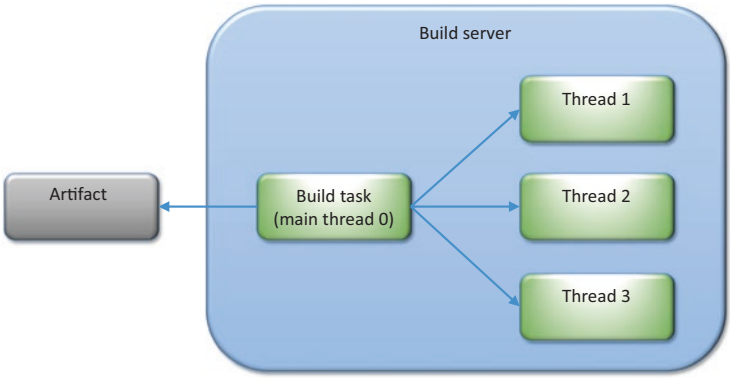


Figure 4-31. *Multithreaded build*

- *Distributed builds*: Multithreaded builds make use of a multicore architecture, spreading compilation over multiple cores in the CPU. But of course, there are limits to the number of cores on one server. A distributed build takes the parallel build concept a step further and distributes the compilation over multiple servers. Even a combination of multithreaded distributed builds is possible, allowing for massive parallel scaling. However, a word of caution is in place if distributed builds are used. Files must be moved around over the network, which costs time. Small projects therefore hardly benefit from distributed builds and might even build faster on just one server. Distributed builds can be considered for large-scale projects.

The principle of a distributed build is that the build of one artifact is split into small individual subtasks, each executed on a different server. This is not the same as an offloaded build, in which the build of one artifact as a whole is offloaded to a separate build server (see the next paragraph). This makes distributed builds more complex in nature than offloaded builds. See Figure 4-32.

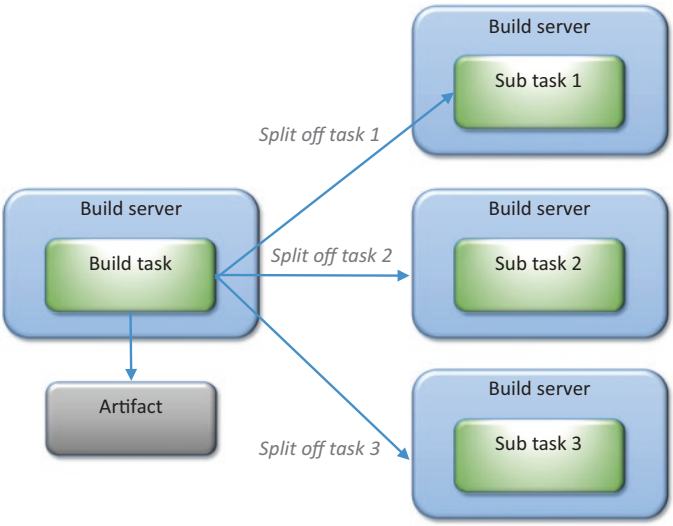


Figure 4-32. *Distributed build*

- *Offloaded builds*: Most ALM/integration platforms provide the means to offload a build to a separate server. The build task that creates an artifact is executed on a designated server or container (sometimes called a *node* or *agent*), depending on the platform. This releases the burden of the main server of the ALM/integration platform and enables parallel builds (of different artifacts). As explained, an offloaded build is

not the same as a distributed build in which the build task of an artifact is split into subtasks, each executed on a different server. This means that the build tool can be simpler and does not need to support a distributed build option. See Figure 4-33.

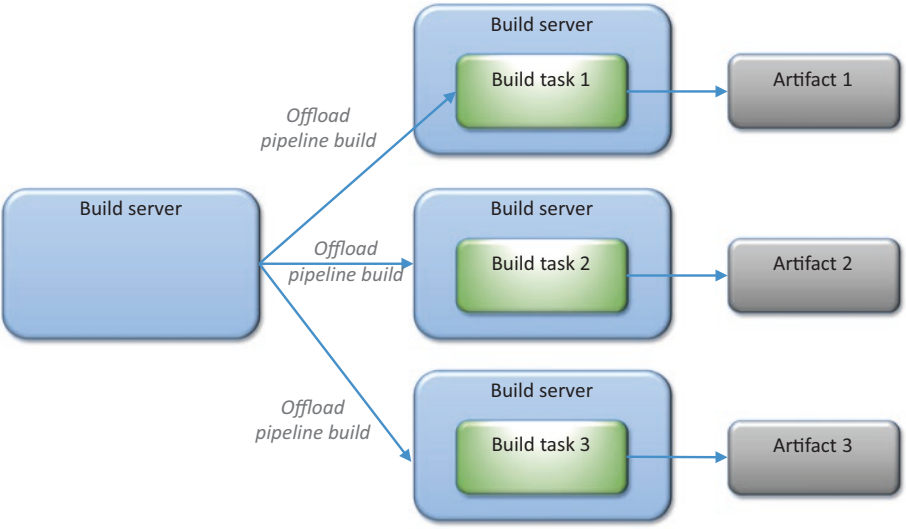


Figure 4-33. *Offloaded build*

Pipeline Caching

Deciding on a strategy to reduce the build time involves not only the execution of a build in terms of CPU usage, but also I/O and networking are big factors to take into account. External libraries used to build an artifact may be retrieved from a location not close to the ALM/integration platform, for example, Maven libraries from Maven Central, Docker images from Docker Hub, and .NET packages from NuGet. Downloading them from these external locations adds a lot of time to a build task.

Some build tools or ALM/integration platforms themselves support caching of files. When a pipeline runs for the very first time, the external files are downloaded, and the cache is created. This cache is stored locally, "near" the pipeline, and is retrieved again in every new pipeline run. The time to retrieve the cache is much lesser, though. This type of caching is also called *pipeline caching* or *remote caching*. Applying caching to a pipeline can decrease the build execution time by 50 percent or more. It is highly recommended to use caching in a pipeline.

Note Caching is used not only for external libraries but also for incremental builds. Compiled files created in an earlier pipeline run are stored in a cache. A new pipeline run will look into that cache first before a source code file is recompiled. Another benefit of caching is that it becomes possible to apply restricted access policies to a cache and block it for other pipelines.

Build Targets

In addition to build time, there are other factors to take into account when a build strategy is defined. Consider the target environment. Some target environments require the creation of certain types of artifacts, such as a Spring Boot JAR or a Docker container but also impose some constraints on these artifacts. Take a Kubernetes cluster, a cloud target, or a mobile phone, for example. Artifacts must be limited regarding storage size, memory footprint, or CPU usage. An artifact for an AWS lambda may not exceed a certain file size; it must have a fast startup time, and memory consumption must be minimized. So, do not focus only on build time when defining a build strategy, but also take the target environment and artifact constraints into account. Tools such as Quarkus, Micronaut, and GraalVM are focused on these aspects and produce artifacts optimized for a target environment where these constraints are applicable.

Cross-Platform Builds

There are plenty of situations in which one codebase leads to different artifacts, each specific to a certain target platform or even certain versions of that platform. Think of applications that must be able to run on both Windows and Linux or a mobile app developed for both iOS and Android. The CI pipeline needs to produce multiple types of artifacts, each one dedicated to running on a specific target platform. A nice feature of various CI tools and ALM platforms is the Matrix Build strategy. This allows building several artifacts at once, based on the permutation of different language versions, operation systems, and operating system versions. Only one CI pipeline is required to build all artifacts, although multiple types of build servers/agents could be needed to perform the build for a specific operating system.

The deployment (CD) pipeline is separate for each platform. One deployment pipeline could be dedicated to a Windows environment, while the other pipeline is based on a deployment to Linux. This is an example of a fan-out principle. Fan-out applies to stages, tasks, and pipelines.

Figure 4-34 depicts two target environments. The build/deployment ratio is one-to-many: one continuous integration pipeline and two continuous delivery pipelines.

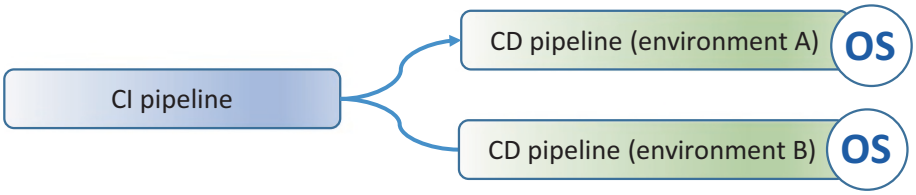


Figure 4-34. *Cross-platform pipelines*

Separation leads to the distribution of activities shown in Figure 4-35.

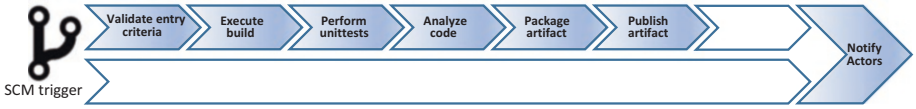


Figure 4-35. *Cross-platform, CI pipeline*

CD pipelines of both environment types are triggered by the same CI pipeline as soon as the CI pipeline is finished. A pipeline-completed trigger can be used for this (see the next chapter for more information about triggers). See Figure 4-36 and Figure 4-37.

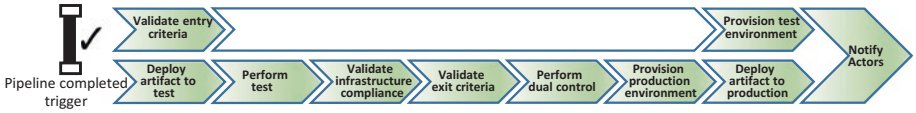


Figure 4-36. *Cross-platform, CD pipeline environment A*

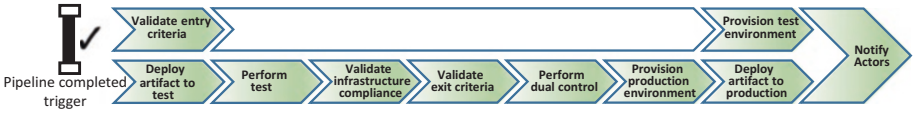


Figure 4-37. *Cross-platform, CD pipeline environment B*

Multiteam Build Strategy

If there is only one production environment to deploy to but multiple teams are developing apps or submodules for that environment, it makes sense to centralize the CD pipeline, managed by one team, while the other teams use their CI pipelines. The reason for just one centralized CD pipeline is to prevent the installation of apps that become rogue in the production environment. Inexperienced teams may introduce vulnerabilities in their apps. These vulnerabilities can be detected by a central CD pipeline. Security checks and stability tests, such as fuzz testing, are added to the CD pipeline to guarantee the stability of apps in the target

environment. The test scope of a team that builds only a small part of the whole system would also test this part in isolation; they will never test how their app behaves as part of a whole system.

Assume a situation that multiple DevOps teams are developing for one product, running in its specific target environment. Each team delivers artifacts that must be assembled into one product. The assembling phase is part of a CD pipeline. A setup to accommodate this is to define CI pipelines managed by individual teams, while the CD pipeline is managed by a central team, which is also responsible for the stability and auditability of the production environment. This setup results in a many-to-one ratio of the number of CI pipelines that perform the build, related to the CD pipeline that executes tests and deploys the artifact to production. See Figure 4-38.

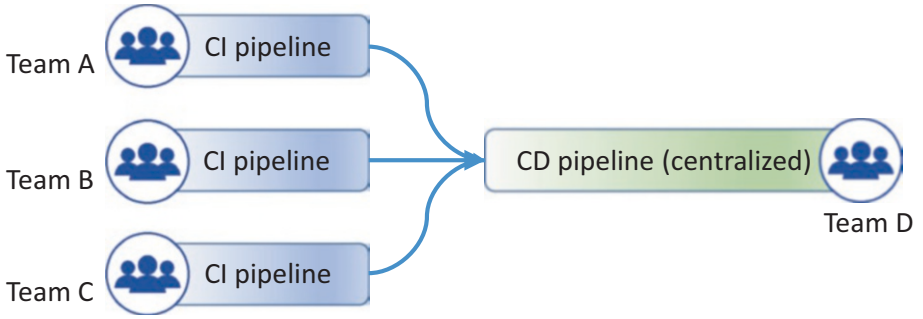


Figure 4-38. *Multiteam build strategy*

Assume your team—team D—is responsible for the design of the centralized CD pipeline. This means you don't even know what other teams—A, B, and C—are doing and what their pipeline looks like. This leads to a "separation of concerns" situation in which one pipeline publishes an artifact to a binary repository, which is fetched by a central CD pipeline, from where it is tested and installed in the central production environment. This separation of responsibilities leads to the distribution of activities shown in Figure 4-39 and Figure 4-40.

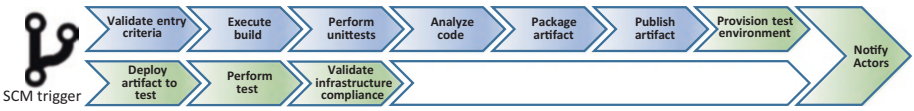


Figure 4-39. *Pipeline teams A, B, and C*

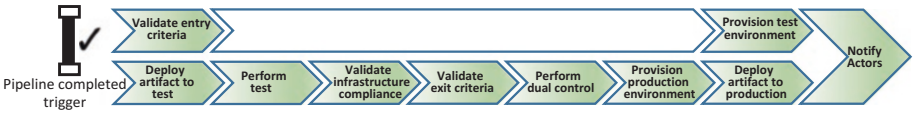


Figure 4-40. *Pipeline team D*

The pipelines of the DevOps teams A, B, and C typically contain all the CI stages. These pipelines also contain test stages to perform integration, system, and contract tests. This gives these teams a feeling of confidence that their app works properly. The pipeline of the central DevOps team (D) is responsible for the target environment and includes all the CD stages. This is also the place where artifacts, produced by the other teams, are integrated and tested as one integral system. The CD pipeline is triggered by all other pipelines, using a pipeline-completed trigger.

Combined, the BPMN workflow model—with collapsed versions of all pipelines—looks like the one in Figure 4-41, in which team A has connected their pipeline to the central CD pipeline through a trigger mechanism. The pipeline of team A submits a trigger, which executes the CD pipeline of team D.

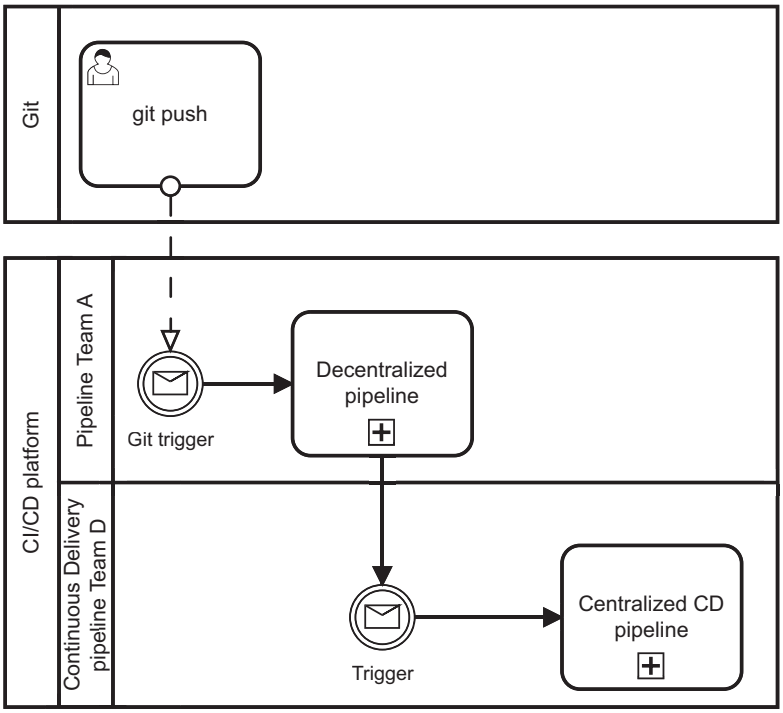


Figure 4-41. *BPMN, multiteam build strategy; CI and CD pipelines combined*

In detail, the design of the CI pipeline of team A could look like Figure 4-42.

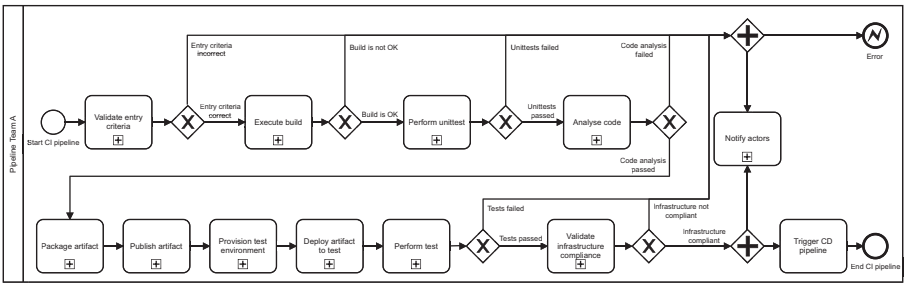


Figure 4-42. *BPMN, multiteam build strategy; CI pipeline team A*

The design of the centralized CD pipeline could look like Figure 4-43.

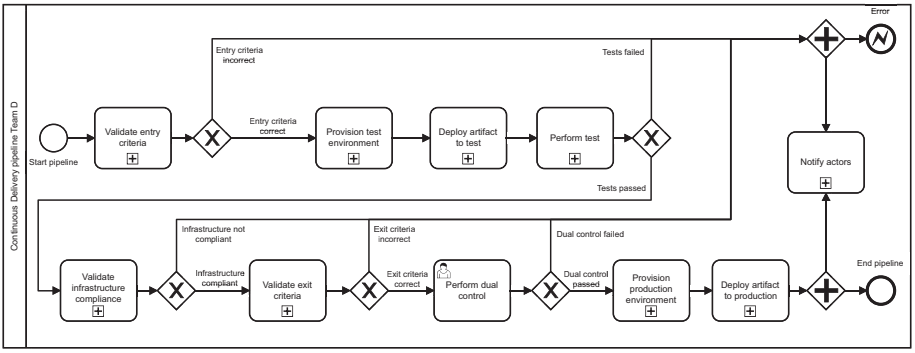


Figure 4-43. *BPMN, multiteam build strategy; centralized CD pipeline team D*

The first stage of the CD pipeline is the *Verify entry criteria* stage. In this stage, the validation of the artifact created by the pipeline of team A is performed to determine whether it meets certain criteria. For example:

- The trigger must supply a valid reference to the artifact.
- The artifact can be retrieved from the binary artifact repository.
- The artifact is a release (candidate), so it must be signed because only signed artifacts are allowed to be installed into production.
- The branch from which the artifact was built is validated. Only artifacts from the main branch are allowed

The multiteam build strategy looks like a simple model at glance, but let's do a small thought experiment. Assume that two teams—A and B—develop, build, and deliver their artifact. The centralized CD pipeline of team D is depending on both artifacts, for whatever reason. It cannot start if one of the artifact versions is not available yet. The problem is, it is not clear which team delivers their artifact first, team A or team B. The central CD pipeline requires a special trigger that depends on two events: pipeline A is finished *and* pipeline B is finished. This trigger mechanism can be considered a type of complex event processing (CEP), which makes it possible to automatically start a pipeline, based on the output of other pipelines. At the moment, the ALM/integration platforms that were investigated do not offer this type of trigger, so if your situation demands a CEP solution, you have to do it yourself, unfortunately.

Test Strategy

The test strategy outlines the testing approach within the software supply chain. Decisions about the order of testing, the fact that some tests run in parallel, the types of tests, which tests are automated, and which are performed manually all contribute to the test workflow. There is no silver bullet on how to design the pipeline flow concerning testing, but there are some typical characteristics of testing that makes certain pipeline flows more logical than others.

A test strategy cannot be discussed without looking at the different types of tests in more detail. Tests come in different flavors, each specialized in a certain area. Some tests focus on functionality, and some on nonfunctional aspects. Also, the scope of tests differs, from narrow-scope tests such as unit tests to broad-scope tests such as chain tests. The question is, how does each type of test impact a pipeline flow? Is there a logical order for all these different test types? Is there a relation between the different test types, and to which extent can these test types be

automated? For the latter question, the testing pyramid described by Mike Cohn in his book *Succeeding with Agile* comes to the rescue (see [4]). But before the relationship between the testing pyramid and pipeline design is handled, here is an overview of possible test types:

- *Unit tests*: Validate the functional behavior of an individual unit of source code by writing unit test cases. Performing unit tests already has a distinct place in the Generic CI/CD Pipeline. Unit tests are executed just after the artifact has been built.
- *Contract tests*: Test the integration between two systems in isolation, mocking the service provider.
- *Integration tests*: Validate the interaction between some components. Where unit tests are performed on individual components, the integration tests are performed on a group of components. Integration tests are functional in nature.
- *System tests*: Validate whether the system as a whole meets the functional and (some) nonfunctional requirements.
- *Regression tests*: Verify that a code change does not impact the existing functionality. Regression tests ensure that the application still performs as expected.
- *Acceptance tests*: Their purpose is to validate whether the system works as expected. This is a formal test because the customer accepts the software if all business requirements are met.
- *UI tests*: These are focused on the user interface of an application. Of course, not all applications have a user interface, so UI testing is very context-dependent.

- *Security tests*: This includes dynamic application security testing (DAST), interactive application security testing (IAST), and penetration testing.
    - Penetration tests are typically performed by an experienced ethical hacker trying to penetrate the system and simulating a cyberattack to validate the systems' security weaknesses. These tests are performed manually.
    - DAST is similar to pentesting, but all tests are automated.
    - IAST tests involve a continuous analysis of a running application often using an agent running on the target environment on which the application is deployed.
- *Preproduction/staging tests*: These validate the provisioning of the infrastructure resources (middleware, databases, etc.) and the deployment and installation of the application artifact on a target environment that is identical to the actual production environment.
- *API tests*: Where contract testing focuses on testing an API in an isolated environment, API tests test the real API. API tests are focused on testing the API contract, its functionality, and its performance. Parts of the API tests are also included in other test types, such as performance tests. It is considered a specific test type because an API is an official contract with other parties, which requires some specific attention.

- *Performance and availability tests*: This includes load tests, stress tests, availability tests, endurance tests, and break tests. Its purpose is to validate nonfunctional requirements related to performance and availability. Tests are executed under heavy load, which is increased until the moment the application breaks. In addition, tests under load run for a long time to validate how the application behaves over time and how stable it is. Endurance tests are typical tests to check whether memory leaks occur after some time.
- *End-to-end tests*: These simulate real user scenarios from beginning to end. It performs the functions within the application that include communication with hardware, databases, file integration, API integration, and messaging with external systems.
- *Disaster tolerance test*: The purpose of disaster tolerance testing is to identify any weaknesses or vulnerabilities in an organization's disaster recovery plan and to ensure that the plan is effective in restoring critical functions and operations as quickly as possible. By conducting regular disaster tolerance tests, organizations can identify and address any issues before they become a real problem.
- *Usability tests*: This is a specialized test type. It is focused on user experience, user-friendliness, efficiency, and accuracy of the application. Also, aspects like cross-browser experience are part of the usability tests. These are manual tests.

The testing pyramid of Mike Cohn distinguishes only a few test types. In Figure 4-44, an attempt is made to map a range of test types to the testing pyramid.

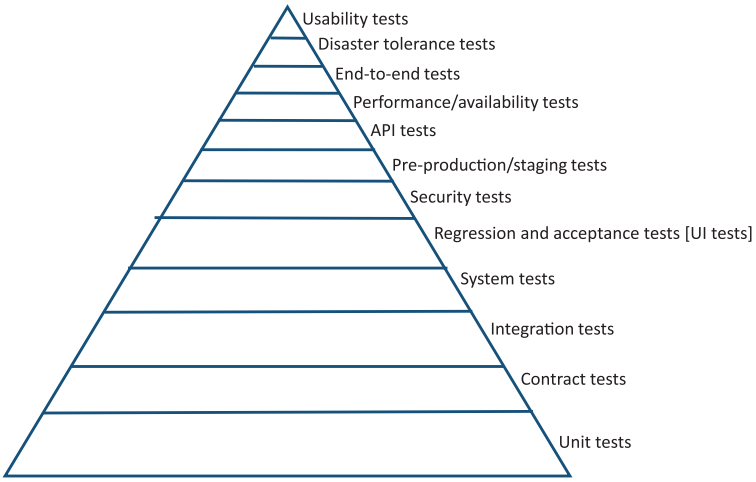


Figure 4-44. *Testing pyramid*

The testing pyramid categorizes these tests. The bottom layer represents quick wins. These form the bulk of tests that are relatively easy to automate. The test types at the top are fewer in number but more difficult to automate and therefore more expensive to automate. The pyramid, therefore, suggests a certain order in which tests should be executed. Consider the *Perform test* stage of the Generic CI/CD Pipeline. The order of test tasks in the *Perform test* stage is directly copied from the testing pyramid (except for unit tests because these are already executed earlier in the flow). This gives an anchor point for the realization of the test flow in a pipeline. See Figure 4-45.

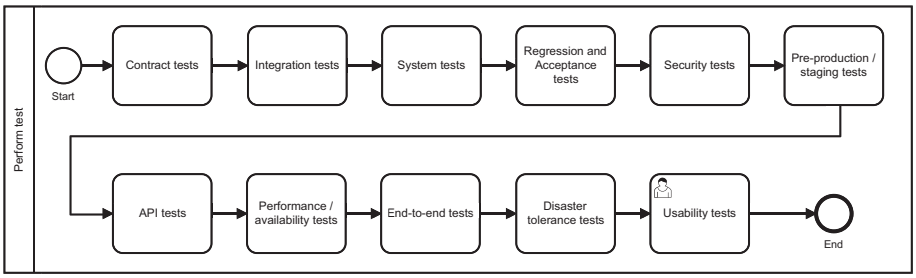


Figure 4-45. *BPMN, ordering individual test tasks based on potency to automate*

This model ranks the tasks only from "relatively easy to automate" to "too difficult to automate." By default, usability and pentests are manual, and as the model shows, all manual tests are executed at the end of the stage. We could leave it to this and conclude that a *Perform test* stage contains these tasks in the proposed sequence.

But this is not the whole story. Besides the distinction between "relatively easy to automate" and "too difficult to automate," there are more test dimensions to consider. Given the five dimensions listed next, which one contributes the most to the order of tests? What dimension is the most important, and which one contributes the least? Let's propose the following order:

- *Automated vs. manual tests*: One of the principles of CI/CD is that all tests must be automated. The next pages will demonstrate what the impact is on the pipeline if manual tests are included in the workflow. A general rule of thumb is that automated tests are executed before manual tests. This is the first dimension to consider.

- *Functional vs. nonfunctional*: Although nonfunctional requirements—including security requirements—are important, the business owner initially focuses on the functional requirements. This argues in favor of placing the functional test tasks in front of nonfunctional test tasks. Note that in cases in which all tests are automated, the order of these tests usually does not matter anymore.

- *Parallel execution vs. sequential execution*: The third dimension determines which tests can be executed in parallel. Running tests in parallel decreases the overall test time and increases fast feedback. Group the automated tasks that can be run in parallel. This is bound by the ability of the ALM/integration platform, to which extent parallelization of test tasks is possible, and this is bound by the capacity of the test environment. Some ALM/integration platforms contain features to order tests automatically, based on historic execution time. This optimizes the overall test time.

If possible, execute manual tests also in parallel. This depends on the capacity of the QA team and test specialists, of course.

- *Manual tests performed by specialists*: Specific test types require specialized test engineers. Pentests and usability tests require certain expertise usually not found in the team itself. So, these people have to be arranged, and because specialists are often hard to allocate, these types of tests must be carefully planned. Within the group of manual tasks, postpone manual test tasks performed by a specialist and first focus on the manual tests that can be performed by the QA team itself.

- *Long execution time vs. short execution time*: Something that usually cannot be designed up front is the fact whether a test task runs short or long. This results in a redesign of the pipeline in a later stage should this situation occur. In that case, the tests with a shorter execution time must be placed in front of the test with a longer execution time, in case not all tests can be parallelized. There is an exception, though. If the execution time of the tests takes a long time—think hours—one must consider isolating this task and excluding it from the main pipeline. In one of the following paragraphs, an example of this situation is given.

Keep the order of these dimensions in mind during the upcoming paragraphs.

Automated vs. Manual Tests

There are two types of tests, the ones that contribute to CI/CD and the ones that block CI/CD. In other words, there is only automated testing and manual testing. Automated testing is repeatable, fast, and reliable, while manual testing is error-prone and time-consuming.

One of CI/CD foundations is that tests are automated, and an effective pipeline does not contain manual testing. However, in practice, manual testing cannot be prevented. As already explained in the requirement "Only allow manual testing if needed," there are several reasons why manual testing is still needed. Here are two examples:

- The QA team has a backlog integrating test cases into the automated test set.
- Execution of rare tests is too expensive to automate and is performed manually.

The problem with manual testing is that it blocks a pipeline, causing the pipeline to become orphaned. This means that the pipeline is not completely finished. Some tasks are yet to be done, and the pipeline is just waiting for all tasks to be completed. Consider a case in which the development team merges three finished features back into the trunk. With every merge of a feature to the source management system, a pipeline instance starts, basically resulting in three runs of the pipeline. If the pipeline contains a manual test task, it halts until the manual test task is finished. If the QA team has a backlog in executing manual tests, these three pipeline instances wait in the *Perform test* stage, waiting for a test engineer to execute the manual test. The most obvious choice is to test the three features in one go, which corresponds with the latest pipeline run—pipeline instance C in Figure 4-46—because it covers all three features. The other two pipeline instances—A and B—are dangling and must be stopped manually.

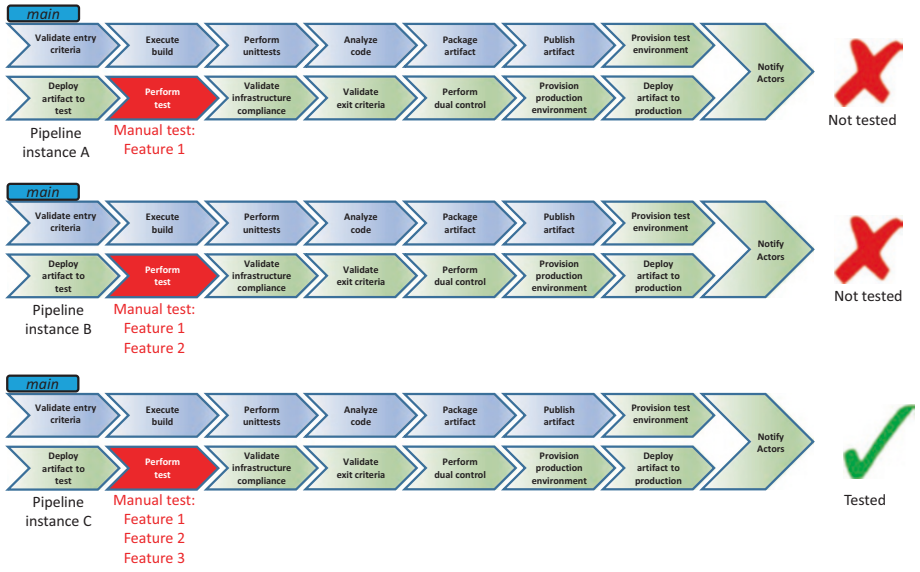


Figure 4-46. *Two dangling pipeline instances*

Dangling pipelines are not a big problem if a manual test can be performed very fast and the QA team can keep up with the speed of development. A manual test that takes up an hour doesn't have to be a big issue. The pipeline is only temporarily blocked. But this is often not the case, and manual tests are waiting in the queue for a long time, while the number of pipeline instances grows.

A question you might ask is what it exactly means when a pipeline is being blocked by a manual test. A running pipeline itself does not include any manual test. Manual testing is something performed outside the pipeline. But, if a QA team is finished with its test, the results need to be registered and signed off. Registration is required to prove that the test was executed and record who tested it. The registration is a manual activity in the pipeline. The test engineer fills in the test results and the location of the test report in an edit box in the pipeline and clicks Confirm. From that moment the pipeline continues.

So, what are the options to streamline the pipeline flow and prevent dangling pipeline instances? First, it must be clear that changing the pipeline flow does not solve the fact that manual tests are still pending, but it is a matter of cosmetics to isolate this manual test stage from the main flow. A few options are at your disposal.

- *Park the manual test stage*: You can split off manual testing from the *Perform test* stage and make it a separate stage. The positioning in the pipeline flow is arranged in such a way that manual testing is "parked." It is still actively waiting to be executed, but it does not block the pipeline anymore. Figure 4-47 shows that the stages *Validate infrastructure compliance* and *Validate exit criteria* are executed because manual testing has been moved to a sidetrack (however, the pipeline will stop before the *Perform dual control* stage, as this involves a manual activity).

- Note that the Generic CI/CD Pipeline also includes stages to perform dual control and deployment to the production environment. The dual control stage is also a blocking stage, so the gain by parking the manual tests in the pipeline flow is limited.
- A parked manual test stage looks like Figure 4-47 in BPMN notation; for convenience, the CI stages are combined into one subprocess called CI stages.

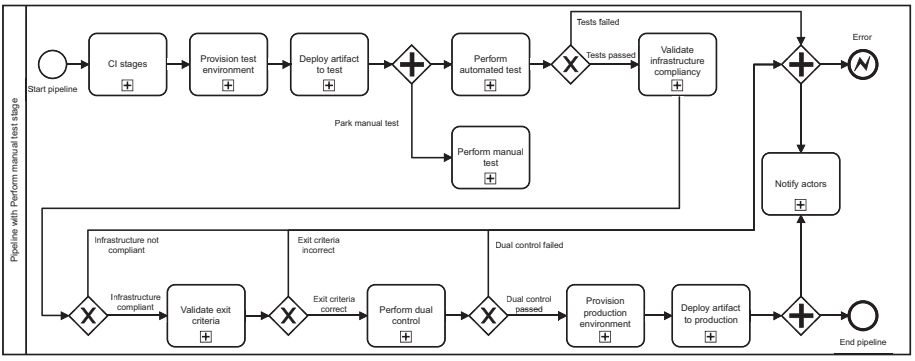


Figure 4-47. *BPMN, parking manual test*

- *Split the pipeline*: Parking the *Perform manual test* stage still results in pipelines with a dangling stage. Splitting the pipeline is an alternative to resolve this. Right in the middle of the *Perform test* stage, between the automated test tasks and the manual test tasks, the pipeline is divided, resulting in two separate pipelines. The first pipeline contains all continuous integration stages to build the artifact and performs all automated tests. The trigger to start this pipeline is a push of the code to the source code management system. After all automated tests are executed, the pipeline ends. This is visualized in Figure 4-48.

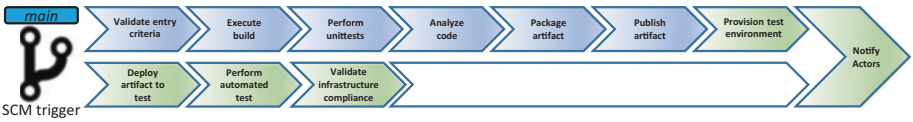


Figure 4-48. *SCM trigger and automated tests*

The second pipeline starts with a manual trigger. The person who started the pipeline is also the one who performs the manual test. The *Deploy artifact to test* stage needs to know which artifact must be deployed, so the manual trigger must include an option to select the already build artifact from the repository or uses the latest version by default. If multiple existing test environments are available, the specific test environment on which the manual test is executed must also be provided as part of the manual trigger. See Figure 4-49.

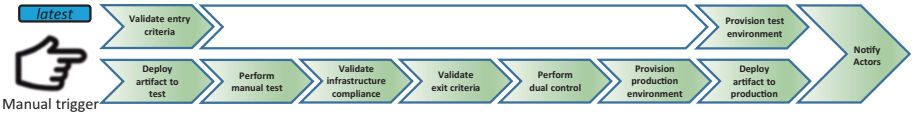


Figure 4-49. *Manual trigger and manual tests*

The benefit of this approach is that the first pipeline executes all stages without being blocked by a manual test. The second pipeline is started only after all manual tests have been executed; otherwise, it makes no sense to start the pipeline in the first place. Using two pipelines like this does not result in dangling pipelines.

- *Separate pipeline for manual tests*: Another approach is to completely isolate the *Perform manual test* stage from the main pipeline and wrap it in a pipeline dedicated to manual tests. This pipeline is either manually triggered or triggered from the main pipeline using a webhook, called by the *Perform test* stage. The

downside of this approach is that this pipeline becomes too isolated from the other pipelines, and the team probably forgets about its existence.

- *Auto-cancel*: Auto-cancel is a nice feature used by a few ALM/integration platforms. The idea behind it is that if a new pipeline instance is started, the already running pipeline instance stops. This means there is always only one pipeline instance active. This prevents multiple dangling pipelines, and it is clear which pipeline run is the most recent one. Consider Figure 4-46. If an auto-cancel option would have been activated, pipeline instances A and B are stopped and only pipeline instance C is active. For the manual test engineer, only the active pipeline instance C is important, and they can ignore pipeline instances A and B.

Functional vs. Nonfunctional Tests

The test types listed previously can be divided into functional and nonfunctional tests.

Functional tests

- Unit tests
- Contract tests
- Integration tests
- System tests
- Acceptance tests
- Regression tests
- UI testing
- API tests

- End-to-end test
- Usability tests

Nonfunctional tests

- Security tests
  - Penetration tests
  - DAST tests
  - IAST tests
- Preproduction/staging tests
- Performance tests
- Disaster tolerance tests

The following rule applies: functional tests are executed before nonfunctional tests. Applying this rule to the sequence defined by the testing pyramid, the tasks are rearranged a bit. Within the automated tests, the nonfunctional tests are positioned at the back of the automated tests. Within the manual tests—notice that more tests in the model are marked as manual—the nonfunctional manual tests are positioned at the back of the pipeline. This results in the model shown in Figure 4-50.

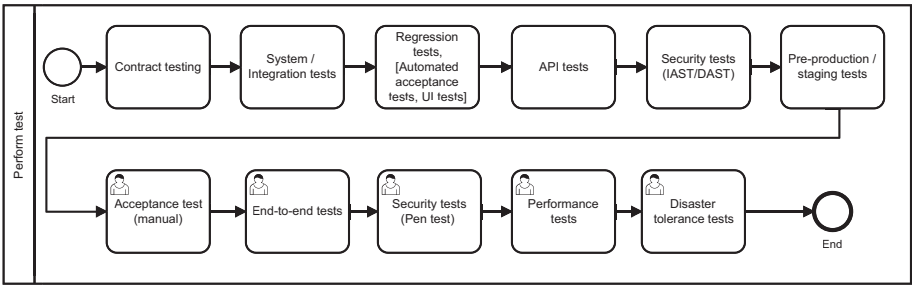


Figure 4-50. *BPMN, ordering test tasks from functional to nonfunctional*

The automated security tests (IAST/DAST) and the reproduction/staging tests are nonfunctional tests and positioned further to the back of the automated test tasks. The same applies to the manual security test (pentest), performance tests, and disaster tolerance tests. They close the rank of the pipeline.

Parallel Execution vs. Sequential Execution

The outcome of one test type should not be the starting point of another test type. Different tests must be executed in isolation, and the configuration of the data of a test is part of the test. The precondition of a test consists of installing certain data files to a file system, prefilling database tables, or preparing a Docker container in which the test is executed. Whether ephemeral test environments or fixed test environments are used, the principle to perform tests independently remains. If tests are executed in parallel, certain constraints are applicable; this applies to both manual and automated tests.

- Tests running in parallel should not interfere. If this happens, some of the tests must be executed on another test environment.
- The number of available test environments could become a bottleneck. Test environments are either fixed test environments, ephemeral test environments created through infrastructure as code, or Docker containers in which tests are executed (e.g., using something like `testcontainers.org`). In the case of Docker containers, the runtime environment of the Docker containers must be sufficiently scaled.
- The ALM/integration platform must support enough parallelized tasks (or jobs). In the case of SaaS platforms, you pay extra for each additional parallel job.

In the case of manual tests, another constraint applies:

- The availability of the test engineers is important, either a test engineer from the QA team or a specialized test engineer. If manual tests are performed in parallel, a suitable balance must be found in the number of test types that can be executed in parallel and the (human) resource capacity.

Going back to our model, some more information needs to be clear about the availability of test environments and test engineers. Assume the following conditions:

`Conditions:`

- *There is no maximum to the number of test environments and parallel jobs in the ALM/integration platform.*
- *The QA team consists of only two test engineers who can perform manual tests in parallel.*

Given these preconditions, the model has been adjusted again, as shown in Figure 4-51.

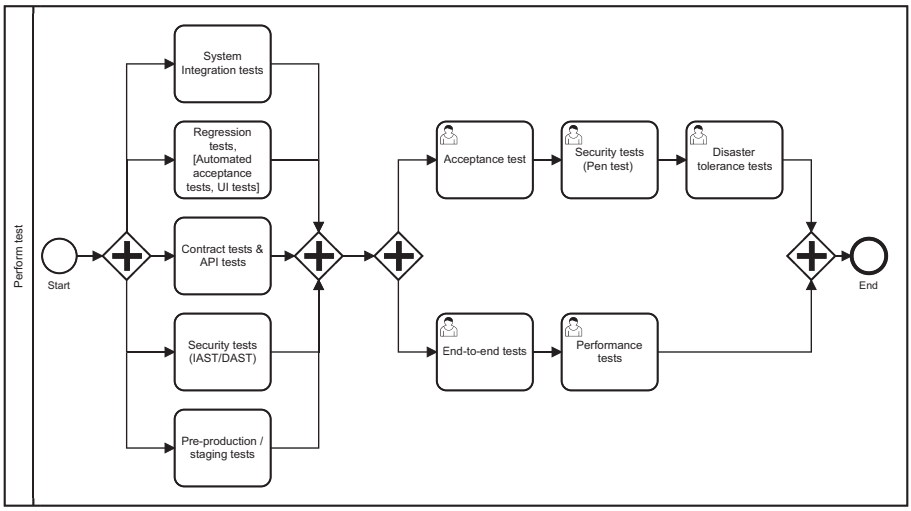


Figure 4-51. *BPMN, applying parallelization*

All automated tests run in parallel. For convenience, the API tests are combined with the contract tests. The difference between functional and nonfunctional tests does not matter anymore in the case of parallel tests. The manual tests are also parallelized. Given that the QA team has only two test engineers, two parallel lanes are defined. The security pentest is positioned a bit arbitrarily because often this expertise is not present within a DevOps QA team. That is solved in the next paragraph.

Note The model defines the different manual tasks as individual tasks and even takes the size of the QA team into account. Why not model this as just one task called "Perform manual tests"? This is possible, but it does not reflect the actual flow. There are different types of manual tests, and by making them discrete in the model, it becomes explicit that there are different test types to be dealt with. This is a matter of taste, of course. You decide whether you want to model this explicitly or not.

Manual Tests Performed by Specialists

The security pentest is performed by a specialist, a cyberspecialist. In Figure 4-51, it looks like it is just executed after the Acceptance test. That will probably not be the case. The pentest can probably be executed only if the application is stable and tested thoroughly. The pentest task must therefore be separated from the other manual tests, as shown in Figure 4-52.

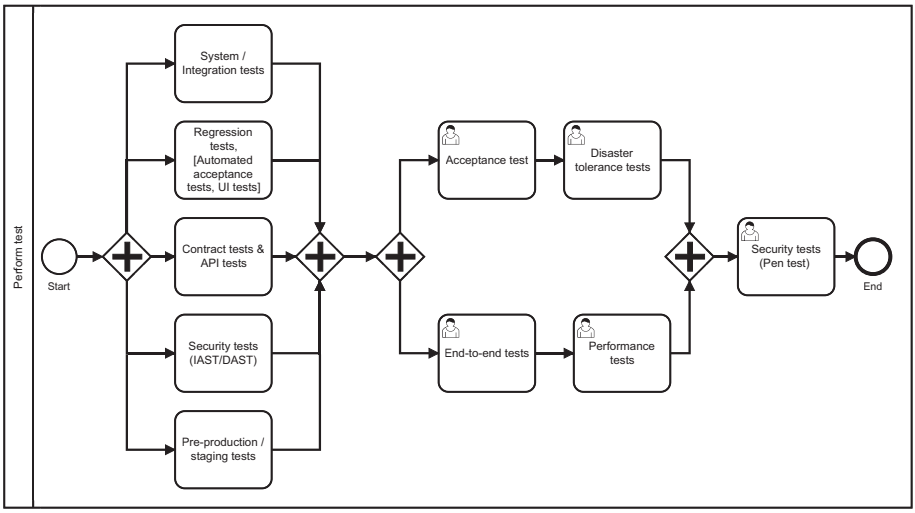


Figure 4-52. *Isolating the specialized test task*

Positioning the pentest in parallel to the other manual tests is an option, but this is possible only if the application is stable enough. In this model, the pentest is moved to the back of the manual tests.

Long Execution Time vs. Short Execution Time

Test execution can take a long time. Assume that one of the test tasks lasts for two hours. This means that manual tests have to wait until this automated task is ready. The long-running automated test blocks the pipeline. To solve this, the specific test task is moved to a different pipeline with a scheduled trigger.

`Condition:`

- The Automated Security tests last for 2 hours.

Given this condition, the final model of the *Perform test* stage looks like Figure 4-53.

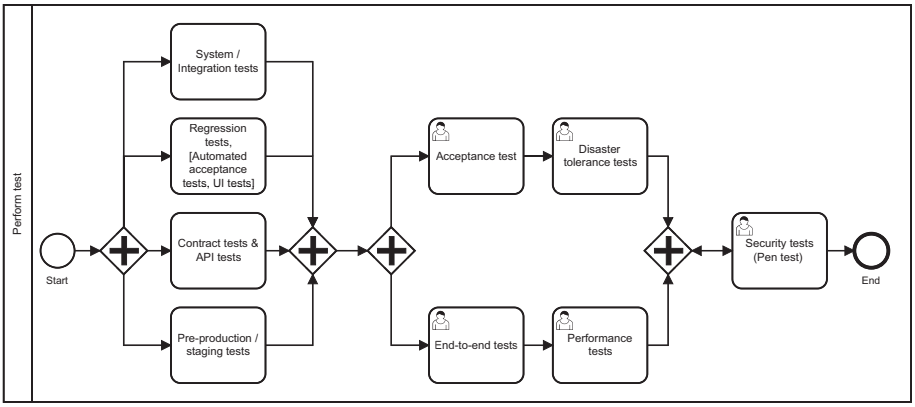


Figure 4-53. *Removed long-lasting test task*

A new pipeline is created, also with a *Perform test* stage, containing the task security tests (IAST and DAST). This pipeline is triggered using a schedule (e.g., starting every evening). Make sure that checking the results of this pipeline is part of the team's workflow, as shown in Figure 4-54.

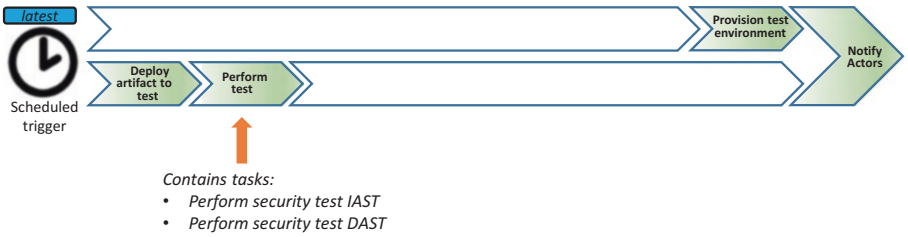


Figure 4-54. *Isolating long-lasting tests in a separate pipeline*

Note The different test models shown just represent an example case. Depending on the context of the tested application, certain test types are not applicable or are combined, resulting in fewer tasks.

Release Strategy

Branching strategy, deployment strategy (which is discussed in the next paragraph), and release strategy sometimes cause confusion, and people tend to mix them up. Let's clarify these concepts.

- *Branching strategy* involves the process of bringing a business feature to the main branch (or to a release branch), with the intention to deploy it to production.
- *Deployment strategy* defines how the artifact is deployed to production. The availability classification of the application is the main driver of the deployment strategy. If downtime is allowed during deployment, a different strategy is chosen compared to a case in which the application must be available 24/7.

- The *release strategy* mainly deals with the moment a deployment to production happens. This can range from a developer pushing code to a repository that is put into production within 15 minutes, to a major release that is deployed after a few months. There may be good reasons to wait for a longer time, for example, if the product you deliver needs to go through formal procedures before it is allowed to be released. The release cycle of the Java JDK, for example, is six months. The team chooses the type of release strategy that fits best in their situation.

Road Map–Based Release

For lack of a better term, a road map–based release seems the best name to reflect the strategy, in which a product owner plots business features on a road map that are linked to a release calendar. This can be a very useful release strategy if, for example, the product road map is aligned with a marketing plan to ensure that marketing efforts are closely tied to the product development process and focused on promoting features and capabilities that are being released in a particular time frame.

The time between each production release is not fixed. The road map may contain two releases that need to be deployed within one month, with a gap of two months until the third release takes place. During development, the team can still practice the principles of continuous integration and continuous delivery, keeping the main branch in a production-ready state. Continuous delivery does not state that every commit to the mainline also has to be deployed to production immediately.

This strategy results in two pipelines: a primary pipeline and a production deployment pipeline (see Figure 4-55).

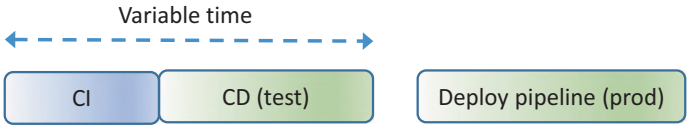


Figure 4-55. *Road map–based release*

The primary pipeline contains all stages, except those related to the production deployment. The stages *Validate exit criteria*, *Perform dual control*, *Provision production environment*, and *Deploy artifact to production* are part of a separate deployment pipeline. These stages are decoupled from the primary pipeline because otherwise it would result in a lot of orphaned pipeline instances. After all, the moment to deploy to production has not been reached.

Separation results in the two pipelines, as shown in Figure 4-56 and Figure 4-57.

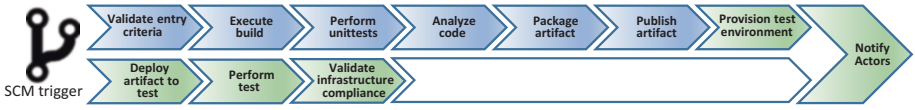


Figure 4-56. *Road map–based release, primary pipeline*

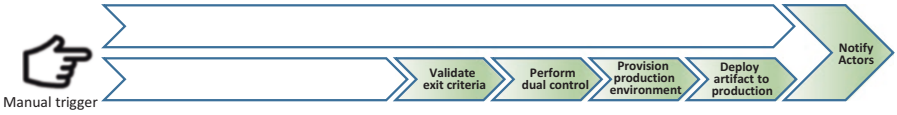


Figure 4-57. *Road map–based release, production release pipeline*

The production release pipeline is manually triggered because the release date varies. The BPMN model of a road map–based release—containing only automated tests—looks like Figure 4-58.

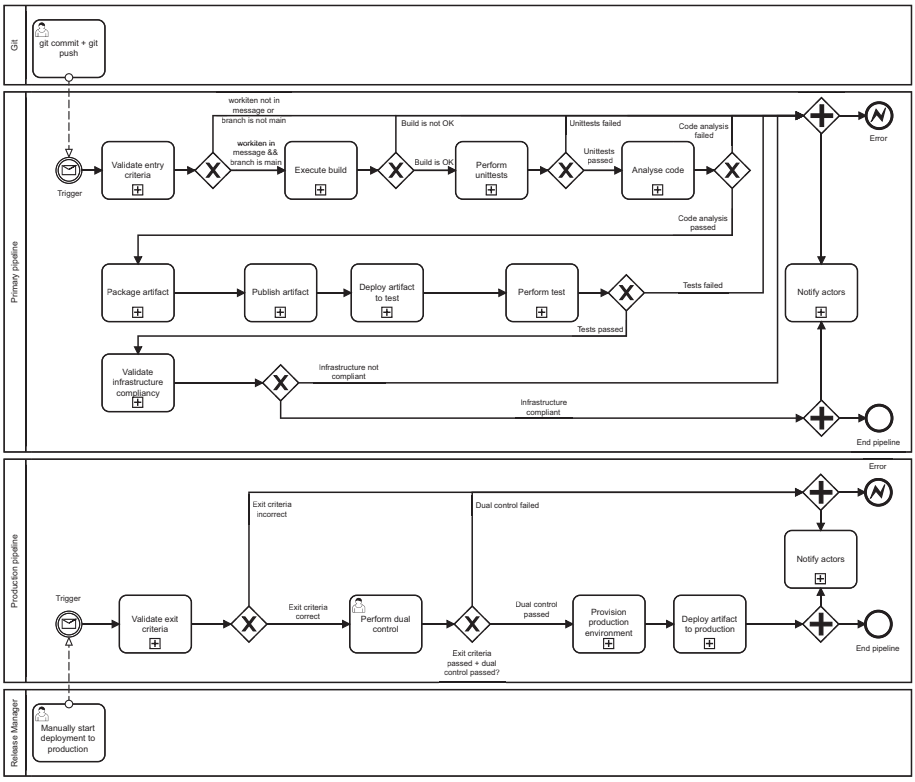


Figure 4-58. *BPMN, road map–based release*

Timeboxed Release

Sometimes, there are valid reasons to deploy to production at regular intervals. A release is timeboxed, meaning that features are added until the end of the timebox has been reached and the deployment to production is performed. A timebox is, for example, a Scrum sprint in which the release is deployed at the end of each sprint. In his blog, Martin Fowler calls this a *release train*. The train arrives and leaves at the scheduled times. When the train leaves the station, all features that stepped into the train go to production (see [27]). See Figure 4-59.

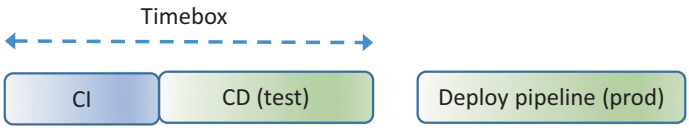


Figure 4-59. *Timeboxed release*

This results in two pipelines: a primary pipeline and a production deployment pipeline. See Figure 4-60 and Figure 4-61.

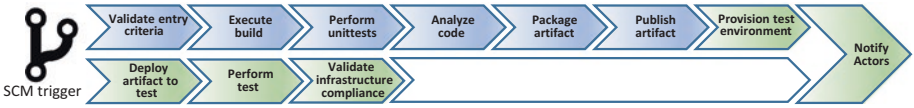


Figure 4-60. *Timeboxed release, primary pipeline*

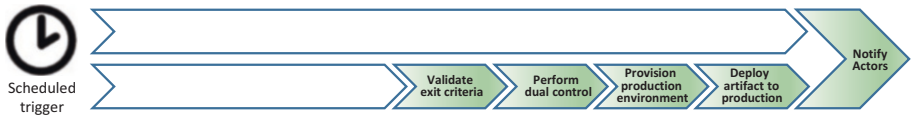


Figure 4-61. *Timeboxed release, production release pipeline*

This strategy looks similar to the road map–based release strategy with the exception that the intervals between the releases are fixed, and the production pipeline is triggered using a schedule.

Note Timeboxes are concatenated. If a feature misses the deadline of a timebox, it is released as part of the next timebox. A variation on this strategy consists of overlapping timeboxes. The next timebox does not start if the previous one has ended but starts halfway through the previous timebox. This allows a feature to be released a bit earlier.

Regular Release

A regular release means that each business feature committed to the mainline is deployed to production as soon as possible. This type of release is possible only if the mainline is kept in a state from where it is possible to deploy to production at any given moment (this is an important continuous delivery principle). This is also possible in the two previous release strategies, but the difference is, in the case of regular releases, deployments to production are done more often, not once per two weeks, but maybe once a day or even multiple times a day. In this strategy, just one pipeline is involved, containing all stages. See Figure 4-62.

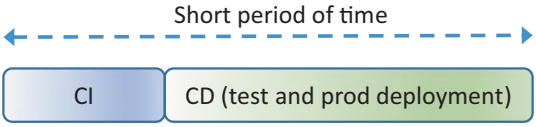


Figure 4-62. *Regular release*

Unlike the road map–based strategy, the deployment in a regular release pipeline is not triggered manually. In a road map–based release, the system owner actively has to start a deployment pipeline and selects the release version they want to deploy to production. In the case of a regular release, the process is automatically triggered by an SCM event. The application is built and tested until the pipeline waits for the system owner to approve the deployment.

A side effect of a regular release is that the number of pipeline instances can start to queue if a lot of business features are added to the mainline in a short time. The system owner probably does not like to approve multiple times a day and approves only the latest release version. Older pipeline instances keep "dangling" in the queue. These older pipeline instances must never be approved anymore; otherwise, this

would result in the deployment of an older release version. The pipeline should always check the release version[9] to mitigate this risk. A pipeline used in a regular release contains all stages, as shown in Figure 4-63.

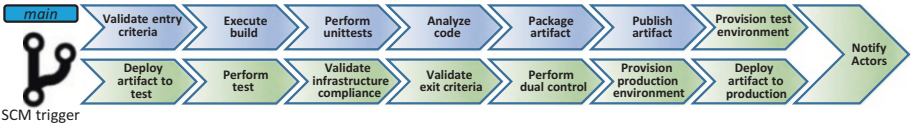


Figure 4-63. *Regular release pipeline*

Continuous Deployment

Continuous deployment is a "hands-off" process in which the deployment to production does not pass a manual dual control stage. This means that if a developer pushes the code to the main branch, the pipeline performs all stages without manual interference, including deployment to production. This results in a pipeline that resembles the Generic CI/CD Pipeline but without the *Perform dual control* stage. See Figure 4-64.

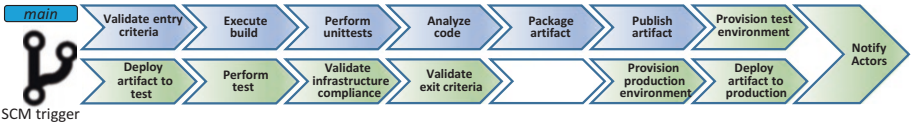


Figure 4-64. *Continuous deployment*

Feature Management–Based Release

A feature management–based release is not a release strategy in itself, but an approach on top of an existing release strategy. Feature management involves hiding business functionality using feature toggles. This makes it possible to release small increments of a business function to production,

[9] The release version in production must always be lower than the deployed release version.

without it being activated. Products like Unleash and LaunchDarkly position themselves in this segment. Feature management can be combined with all branching and release strategies.

Production Deployment Strategy

On an abstract level, deployment to production is depicted as one stage in the pipeline flow, but this stage is sort of an iceberg; it looks simple at a glance, but it is more complicated when zooming in.

The simplest version of deployment is just to overwrite existing files and restart the application. If a database is involved, it is a bit more complicated; a simple deployment already involves creating, altering, and/or dropping tables. In that case, the application has downtime, but if this is acceptable, there is no incentive to implement a more sophisticated deployment mechanism.

But again, real life often poses certain requirements on the system. If downtime is not allowed or must be kept minimal, deployment becomes more complex. And maybe the business organization wants to see whether a certain change in functionality is better received by the target audience than the existing functionality. Having two flavors of this functionality in production and measuring and comparing the performance of the two also poses extra requirements, resulting in a different deployment strategy. Some of the most common deployment strategies are handled in the next paragraphs.

Re-create Deployment

The re-create deployment is best illustrated by an example. See Figure 4-65.

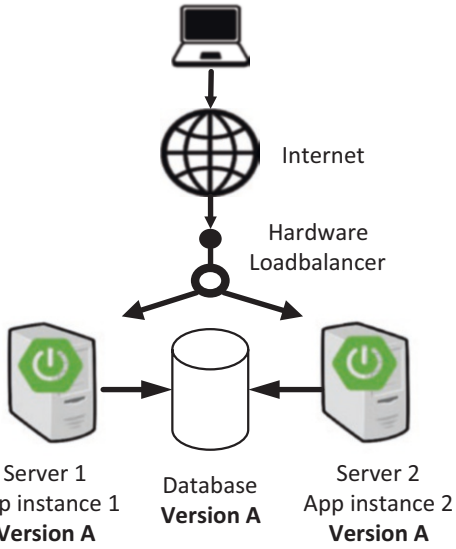


Figure 4-65. *Re-create deployment setup*

Example:

Assume the application is a runnable—Spring Boot—jar, deployed on two Linux servers. The application runs as a Linux service and receives HTTP(S) requests from clients. Communication takes place over the public Internet. Server-side load balancing is performed using a hardware load balancer (for convenience, in this case, there is no client-side load balancing applied). The load balancer redirects the requests to the application instances. Both application instances are connected to a SQL database.

Before the new version (version B) of the application is deployed and the database updates are applied, all communication to both servers is stopped. The nodes (servers) in the load balancer pool are set to "disabled" or "maintenance," which prevents new connections with the servers. HTTP traffic to the servers bleeds dry, and after some time, the applications on the servers do not receive requests anymore. The load balancer reroutes requests from the Internet to a maintenance page informing the client that the application is in maintenance and not available.

This is the moment to stop the applications (stop the Linux service) and overwrite them with the new version. Version A is still installed on each server, but it is replaced by the new version, version B. The database is updated to the new version by running a SQL script that creates, alters, and/or drops tables, depending on the changes of the particular version. After the applications and database are updated, both applications on the servers are started again. If they pass the boot sequence, the nodes in the load balancer pool can be enabled again, and the applications become available. See Figure 4-66.

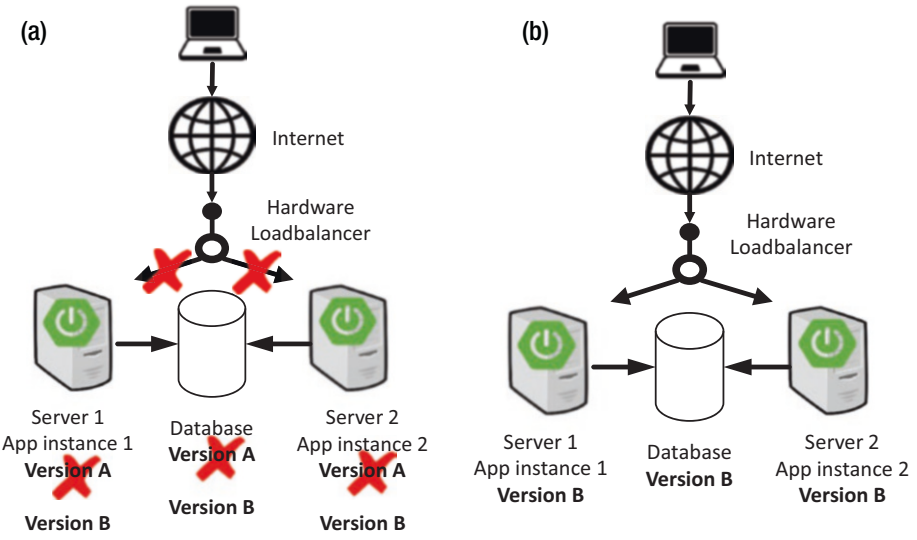


Figure 4-66. *(a) Installing version B. (b) Version B installed and available*

This strategy involves a couple of tasks. All tasks can be automated. See Table 4-4 and Figure 4-67.

Table 4-4. *Re-create Deployment Tasks*

Task	Description
Disable nodes in load balancer pool.	Disable servers 1 and 2 in the load balancer pool.
Wait for a short period (until no requests received).	Wait until the load balancer does not forward any request to the Linux services and the current request is completely processed.
Stop the Linux services on servers 1 and 2.	The Spring Boot app runs as a Linux service. Stop the service using `sudo systemctl myApp stop`.
Copy the JAR file with the new version to the target environment.	Retrieve all artifacts from the artifact repository and copy the application JAR to the target environment.
Copy the DB script to the target environment and execute.	This is the script to migrate from database version A to version B.
Start the Linux services on servers 1 and 2.	Start the Spring Boot app again using `sudo systemctl myApp start`.
Wait for a couple of seconds.	Needed to bootstrap and initialize the apps.
Enable nodes in load balancer pool.	Route request to servers 1 and 2 again.

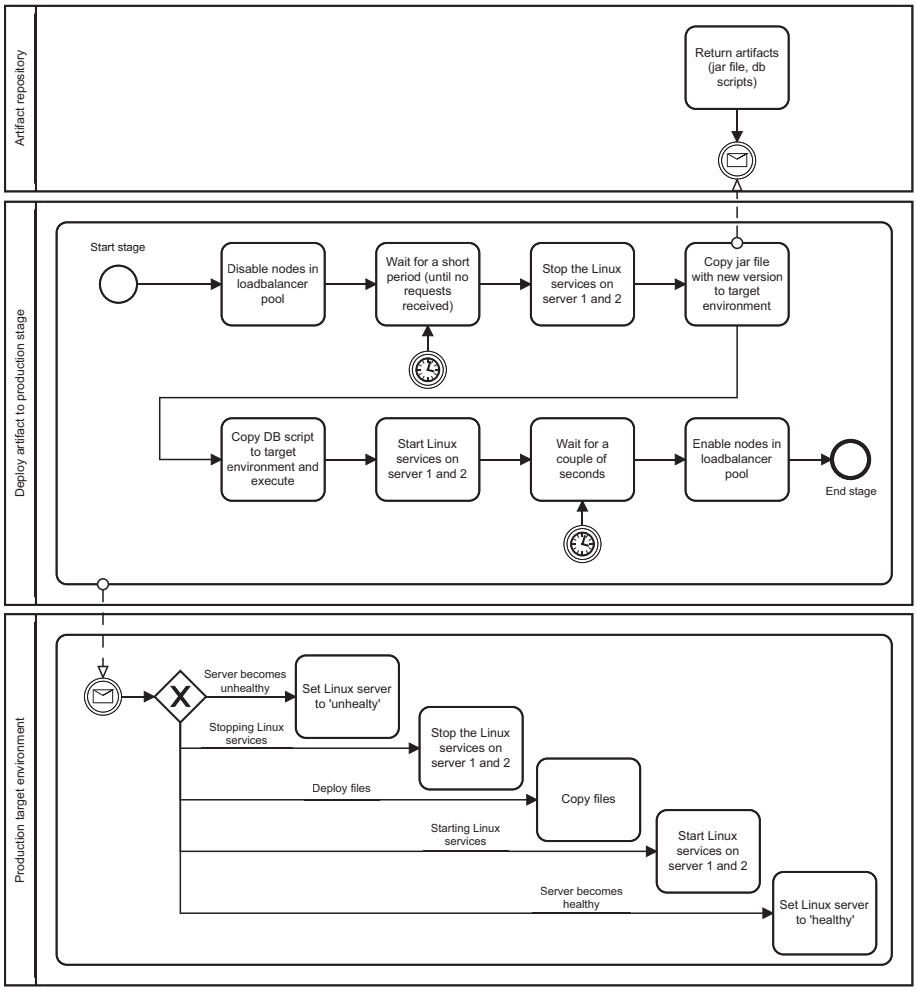

Artifact repository
Return artifacts (jar file, db scripts)
Deploy artifact to production stage
Start stage
Disable nodes in loadbalancer pool
Wait for a short period (until no requests received)
Stop the Linux services on server 1 and 2
Copy jar file with new version to target environment
Copy DB script to target environment and execute
Start Linux services on server 1 and 2
Wait for a couple of seconds
Enable nodes in loadbalancer pool
End stage
Production target environment
Server becomes unhealthy
Set Linux server to 'unhealty'
Stopping Linux services
Stop the Linux services on server 1 and 2
Deploy files
Copy files
Starting Linux services
Start Linux services on server 1 and 2
Server becomes healthy
Set Linux server to 'healthy'

Figure 4-67. *BPMN, re-create deployment tasks*

The model shows how the tasks in the *Deploy to production* stage result in the remote execution of these tasks in the production environment. This also implies the existence of an SSH connection between the ALM/integration platform and the production environment.

Note Disabling/enabling the nodes in the load balancer pool is implemented as a validation performed by the load balancer. The load balancer periodically calls the Linux server with a health request. If the server is in "maintenance," regular requests are not sent to the server anymore, until the load balancer detects that the server is "available" again. Switching between "maintenance" and "available" can easily be implemented on the Linux server.

The re-create deployment strategy is the easiest strategy, but it results in downtime of the application. Other strategies have better ways to reduce or eliminate downtime.

Blue/Green Deployment

In a blue/green deployment strategy, the starting point is an infrastructure with the old version (version A, the blue version) of the application and the database. In parallel, a new infrastructure is built, which has the new version (version B, the green version) installed. The load balancer instantaneously switches from infrastructure A to B, routing the traffic to the new version. If the system has a database, two options are possible.

- The new version of the application can work with the old version of the database.
- The old version of the application can work with the new version of the database.

Often, the first option is not possible because a new version of the application usually requires a database change, specific for the new application version. In the example used in this paragraph, the second option is used. The starting point in this example is a server pool (server

pool A) containing both servers 1 and 2, each running application version A. The loadbalancer spreads all requests over both servers in the pool. See Figure 4-68.

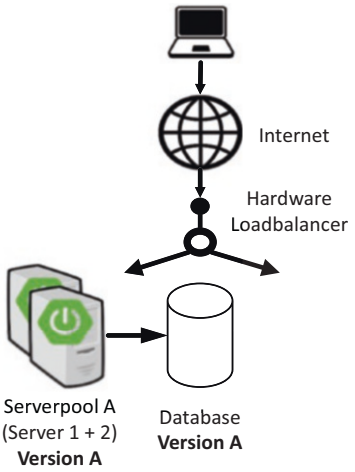


Figure 4-68. *Blue/green deployment, version A installed and available*

The first step in the deployment is to replace the database version from version A to version B. The database script is executed, but because the old version of the application still works with the new version of the database, everything should still be working.[10] The assumption is that the database changes can be performed online, of course. After this has been done, a new infrastructure is built. The new infrastructure contains a server pool (server pool B) with servers 3 and 4. Application version B is installed on both servers, but because no requests are sent to the servers yet, servers 3 and 4 are still idle. See Figure 4-69.

[10] Not all database changes are backward compatible. Sometimes, some additional processing or transformation is required in the database using database triggers, for example.

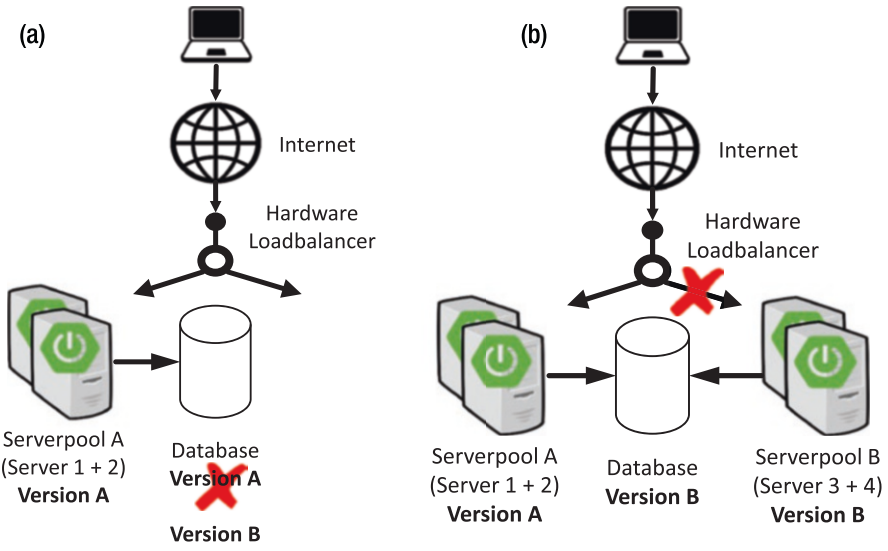


Figure 4-69. *(a) Installing database version B. (b) Second infrastructure with version B installed (but still idle)*

The setup now consists of two server pools, one with application version A and one with application version B. Server pool A is enabled and processes all requests (using database version B). Server pool B is idle. The essence of a blue/green deployment is that the load balancer switches from server pool A to server pool B instantly. After the switch, all requests are sent to the servers in server pool B. Server pool A becomes idle and does not process any new requests anymore. The infrastructure of server pool A can be dismantled and used for other purposes. See Figure 4-70.

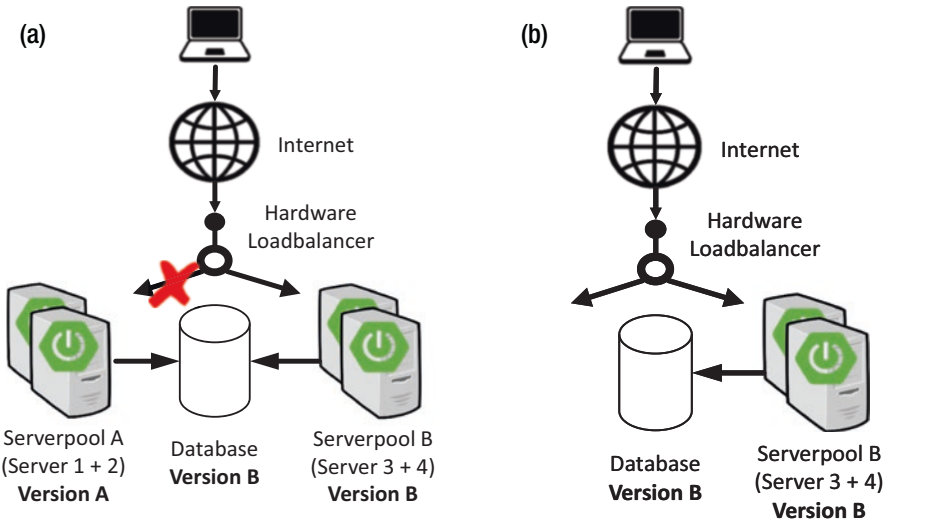


Figure 4-70. *(a) Switch from server pool A to B (server pool A becomes idle). (b) Version B available*

In contradiction with the re-create deployment strategy, always one of the application versions is active. There is no downtime, so all incoming requests are processed by the application. The tasks involved are summarized in Table 4-5. See Figure 4-71.

Table 4-5. *Blue/Green Deployment Tasks*

Task	Description
Provision new infrastructure B.	Create the infrastructure with server pool B. Note that this task is part of the *Provision production environment* stage and not of the *Deploy artifact to production* stage.
Copy the DB script to the target environment and execute.	This is the script to migrate from database version A to version B.

(*continued*)

Table 4-5. (*continued*)

Task	Description
Stop the Linux services in server pool B.	The apps in server pool B are stopped, although this is already the case if the new infrastructure is created.
Copy the JAR file with the new version to the new environment (server pool B).	Retrieve all artifacts from the artifact repository and copy the application JAR to the target environment. This concerns the deployment of the new versions on servers in server pool B.
Start the Linux services in server pool B.	The apps in server pool B are started as a Linux service but do not process any requests yet.
Enable node B in the load balancer nodes pool.	Enable servers 3 and 4 of server pool B in the load balancer nodes pool.
Wait for short period.	To allow bootstrapping and initializing the apps, route traffic to the apps on server pool B. This is the moment both applications A and B are active.
Disable nodes A in the load balancer nodes pool.	Requests to servers 1 and 2 in the server pool A are blocked. From this moment, requests are routed only to servers 3 and 4.
Dismantle the old infrastructure.	Servers in Server pool A are no longer used and can be decommissioned.

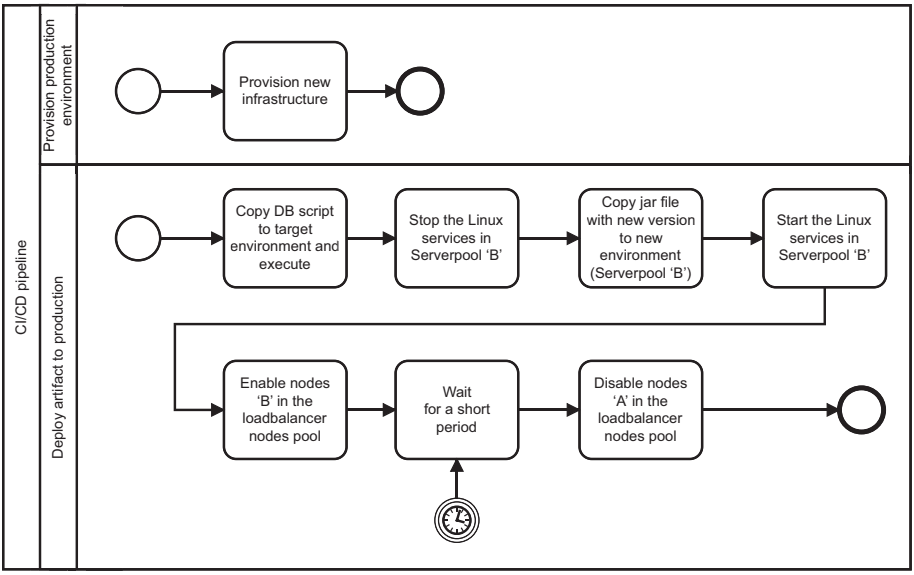

CI/CD pipeline
Provision production environment
Provision new infrastructure
Deploy artifact to production
Copy DB script to target environment and execute
Stop the Linux services in Serverpool 'B'
Copy jar file with new version to new environment (Serverpool 'B')
Start the Linux services in Serverpool 'B'
Enable nodes 'B' in the loadbalancer nodes pool
Wait for a short period
Disable nodes 'A' in the loadbalancer nodes pool

Figure 4-71. *BPMN, blue/green deployment*

For clarity reasons, the BPMN model in Figure 4-71 does not contain a connection between the pipeline and the artifact repository, a connection between the pipeline and the production environment, and the execution of the remote commands in the production environment.

Rolling Update and Canary Deployment

Rolling update deployments differ from blue/green deployments in such a way that blue/green deployment requires two identical infrastructures, while deployment to a new version in a rolling update deployment strategy is done within the current infrastructure on which also the old version runs. In a rolling update deployment strategy, a smaller percentage of the application version is replaced first. If everything looks fine, this percentage is gradually increased.

A canary deployment is similar, with the difference that with a canary deployment, a small percentage of the users are routed to the new version of the application, while the majority of users continue to use the old

version. This allows the new version of the application to be tested in a live environment with a small number of users before being deployed to all users. Because both strategies are similar and primarily focused on testing the stability and reliability of a change, they are used interchangeably.

It is best to demonstrate this strategy using an infrastructure with three servers, each with version A installed. The first step in the deployment is again replacing the database version from version A to version B. The database script is executed, but because the old version of the application still works with the new version of the database, everything should still be working fine.

The next step is to disable server 1 in the load balancer pool. HTTP traffic bleeds dry, and after some time the application on server 1 does not receive requests anymore. All requests from the Internet are routed to servers 2 and 3, which are still active. In the meantime, application version B is deployed to server 1. See Figure 4-72.

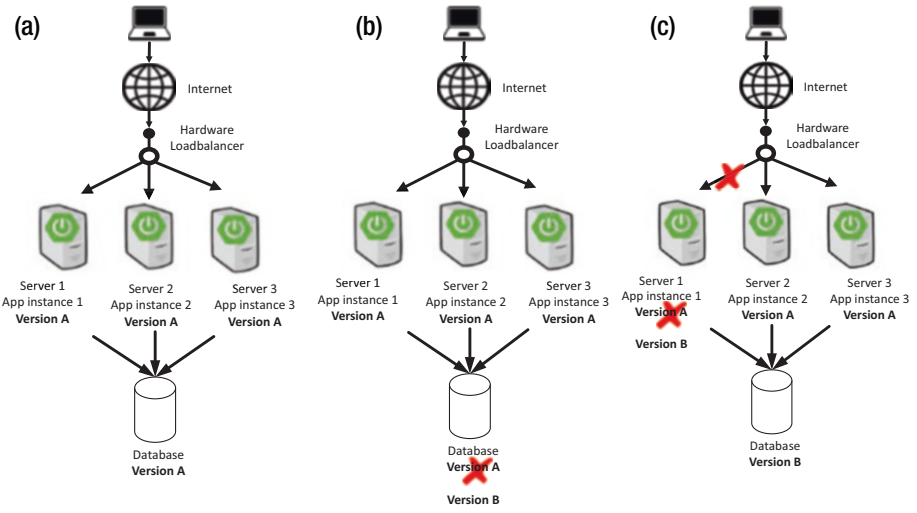


Figure 4-72. *(a) Version A installed. (b) Installing database version B. (c) Version B installed and available on server 1*

After that, server 2 is disabled in the load balancer pool, and server 1 is enabled again. At that moment, server 2 is inactive, and servers 1 and 3 are active. Server 1 serves application version B, while server 3 still serves application version A. Both application versions run at the same time, but because the database is compatible with both application versions, everything works fine. In the meantime, application version B is deployed on server 2.

The next step is to disable server 3 and enable server 2 again. Servers 1 and 2 are active and run application version B, while version B is installed on server 3. The last step is to enable server 3, and from that moment all servers serve application version B. See Figure 4-73.

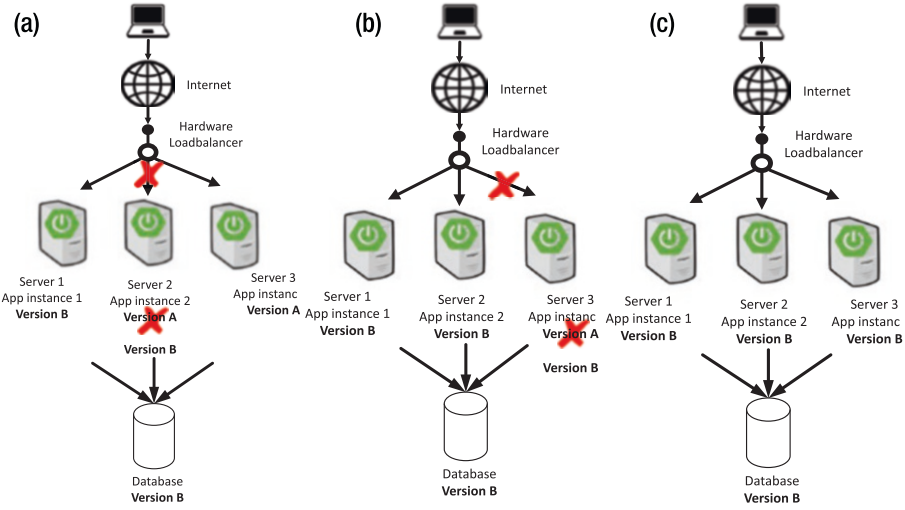


Figure 4-73. *(a) Installing version B on server 2. (b) Installing version B on server 3. (c) Version A completely replaced with Version B*

Table 4-6 summarizes the tasks involved.

Table 4-6. *Rolling Update/Canary Deployment Tasks*

Task	Description
Copy DB script to target environment and execute.	This is the script to migrate from database version A to version B.
Loop; X = Server [1..3].	
Disable node [X] in the load balancer nodes pool.	Block all requests to server [X].
Wait for a short period.	Needed to finish requests that are still processed.
Stop the Linux service on server [X].	
Copy the JAR file with new version to server [X].	Retrieve all artifacts from the artifact repository and copy the application JAR to the target environment.
Start the Linux service on server [X].	Start the Spring Boot app.
Wait for a couple of seconds.	Needed to bootstrap and initialize the app.
Enable node [X] in the load balancer nodes pool.	
X = X + 1.	Increment X to handle the next server.

This results in the BPMN model shown in Figure 4-74. Take note of the repeating task with the intermediate conditional event (iteration). The connection between the pipeline and the artifact repository, the connection between the pipeline and the production environment, and the execution of the remote commands in the production environment are excluded from the model for clarity reasons.

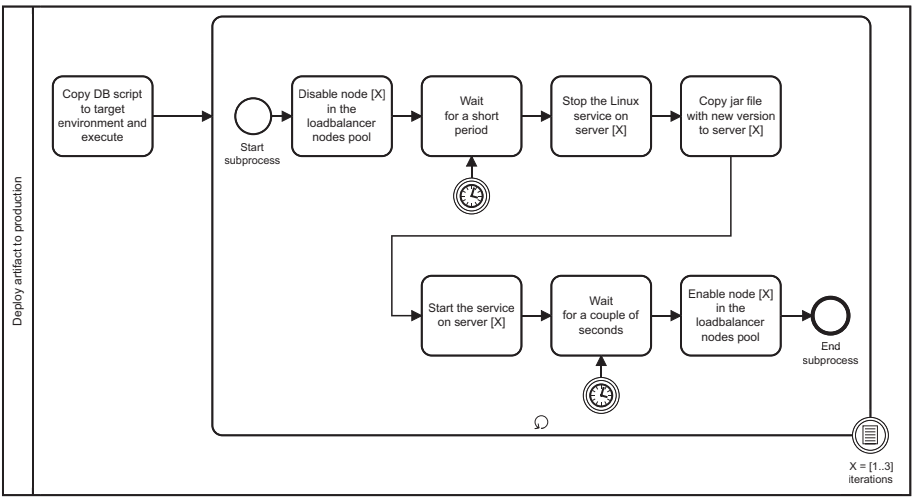

Deploy artifact to production
Copy DB script to target environment and execute
Start subprocess
Disable node [X] in the loadbalancer nodes pool
Wait for a short period
Stop the Linux service on server [X]
Copy jar file with new version to server [X]
Start the service on server [X]
Wait for a couple of seconds
Enable node [X] in the loadbalancer nodes pool
End subprocess
X = [1..3] iterations

Figure 4-74. *Rolling update/canary deployment*

The previous example illustrates a situation with three servers. The processing of application version B is gradually increased in steps by 33⅓ percent. However, in some setups, this may not be sufficient, and other—more controlled—strategies are needed, for example, to increment the processing of the new version in steps of 10 percent. In these situations, the use of specific tooling provides huge benefits. Consider AWS CodeDeploy. It has a feature to deploy using a canary deployment strategy, and instead of programming all the tasks, canary deployment is configured more descriptively by defining a deployment preference type, for example, Canary10Percent10Minutes. This strategy takes care that every 10 minutes, 10 percent of the functionality is deployed, until, after one hour and 40 minutes, all functionality is deployed. If something goes wrong, the deployment is automatically rolled back by AWS CodeDeploy. If possible, use tooling that has deployment strategies built in to avoid programming all the tasks yourself. This also fits well with the CI/CD requirement "Pipeline stages and tasks are orchestrated by the appropriate tool."

Note If an increment of 50 percent is used, the rolling update/canary deployment looks similar to a blue/green deployment, but with a difference. In a blue/green deployment strategy, two identical infrastructures are used. At deployment time, the overall capacity of the infrastructure in case of blue/green deployment is 200 percent, but processing capacity during deployment remains 100 percent (except during the short overlap period in which both application versions are active). Half of the infrastructure is unutilized. In the case of a rolling update/canary deployment with an increment of 50 percent, this results in the current infrastructure temporarily serving two application versions (50-50), and the processing capacity during deployment remains 100 percent. There is no need for a doubling of the infrastructure capacity, so rolling updates are more cost-effective.

A/B Test Strategy

A/B testing is not a real deployment strategy at all. It is a way to test new features in production with a representative user group. In A/B testing, both the old version and the new version are active. Some requests are routed to the old version, and other requests are routed to the new version. A/B testing can be used in combination with both blue/green and rolling update/canary deployment strategies.

Note that by default the result of both deployment strategies is a complete installation of a new version, so the deployment process must be paused along the way if A/B testing has to be squeezed in, having both versions running at the same time. This period can take days or even weeks. After the A/B testing period is finished, the deployment is either continued or rolled back.

Note A/B testing can also be implemented using feature flags.

In the case of blue/green deployment, this means that both the existing and the new infrastructure are active. Requests are partly routed to the old infrastructure, running the blue version, and partly routed to the new infrastructure, running the green version. Using A/B testing in combination with blue/green deployment is more costly because both infrastructures run side-by-side.

If A/B testing is used in combination with the rolling update/canary deployment strategy, the costs are less because the same infrastructure is used, running both the old and new versions. The combination of A/B testing with one of the deployment strategies changes the workflow, though. Let's take the rolling update/canary deployment strategy and combine it with A/B testing. The setup in the example consists of three servers. The deployment stops after version B of the application has been installed on the first server (server 1). The A/B testing period lasts for a month, and after a month, version B is rolled out on the rest of the servers (servers 2 and 3). This means that 33⅓ percent of the requests are processed by application version B, while 66⅔ percent of the requests are handled by application version A.

The assumption is that the pipeline includes some logic and contains a variable with a value indicating the number of servers on which application B is deployed. In the first run of the pipeline, the value of this variable is 1, indicating that application B is installed only on server 1. After a month of A/B testing, the pipeline runs again, with the value of this variable set to 3, indicating that version B of the application is installed on servers 1, 2, and 3. The deployment is idempotent, meaning that if version B is already installed, it is not overwritten with the same version. The list of tasks differs a bit compared to the previous paragraph. See Table 4-7.

Table 4-7. *A/B Testing Tasks*

Task	Description
Copy DB script to target environment and execute.	This is the script to migrate from database version A to version B. This task is idempotent and not executed if the current version is equal to the version to be deployed.
`First execution of the pipeline`: Set variable = 1: A/B testing `Second execution of the pipeline`: Set variable = 3: Complete deployment of B	1: A/B testing (first run of the pipeline). 3: Complete deployment of B (second run of the pipeline).
Loop; X = Server [1..variable]	Skip server [X] if installed version is equal to version to be deployed.
Disable node [X] in the load balancer nodes pool.	Block all requests to server [X].
Wait for a short period.	Needed to finish requests that are still processed.
Copy the JAR file with the new version to server [X].	
Start the service on server [X].	Start the Spring Boot app.
Wait for a couple of seconds.	Needed to bootstrap and initialize the app.
Enable node [X] in the load balancer nodes pool.	
X = X + 1.	Increment X to indicate the next server.

This results in the BPMN model shown in Figure 4-75.

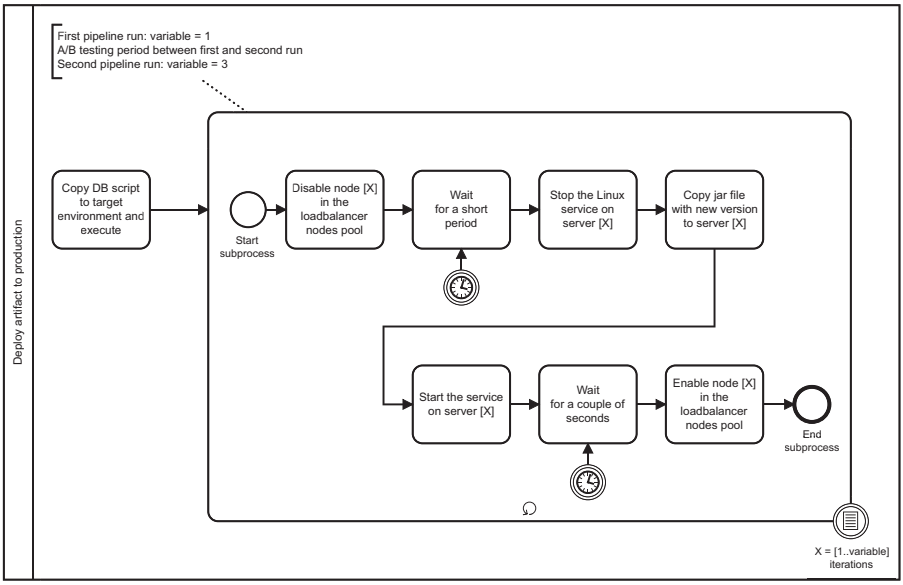

First pipeline run: variable = 1
A/B testing period between first and second run
Second pipeline run: variable = 3
Deploy artifact to production
Copy DB script to target environment and execute
Start subprocess
Disable node [X] in the loadbalancer nodes pool
Wait for a short period
Stop the Linux service on server [X]
Copy jar file with new version to server [X]
Start the service on server [X]
Wait for a couple of seconds
Enable node [X] in the loadbalancer nodes pool
End subprocess
X = [1..variable] iterations

Figure 4-75. *BPMN, A/B testing and deployment strategy*

The BPMN model of the A/B test strategy is similar to the BPMN model of the previous paragraph, with the exception that a variable is introduced for A/B testing to control to which extent version B is deployed.

Other Design Considerations

Here are some other design considerations:

- *Separation of concerns*: Sometimes there are good reasons to decompose a single pipeline into multiple pipelines, each with its specific responsibility. We've seen a few situations in the previous paragraphs, in which a pipeline was split into multiple pipelines. But there are other considerations for distributing responsibilities over multiple pipelines.

- *Role separation*: Developers are focused on creating software and are mainly busy with continuous integration, while Ops engineers have a better understanding of the environments and mainly deal with continuous delivery. In addition, the quality assurance team is dedicated to automating all tests, and external teams may have specific knowledge of certain parts of the infrastructure not managed but used by the DevOps team.

  This knowledge and role separation can also be extrapolated to separating pipelines for specific areas of continuous integration, continuous delivery, quality assurance, and specialized infrastructure managed by another team.

- *Resource constraints*: Teams may be working very actively and pushing a lot of code, which results in queuing of the pipelines because test resources are limited. Decoupling the CD process from the CI process could help. For example, the CD pipeline will be started on a scheduled basis and not after every code push.

- *Carbon dioxide footprint*: Some parts of the pipeline are perhaps very "compute resource" intensive. Source code analysis and automated tests are performed the whole day, sometimes for very small changes that could easily have been combined with other features. This puts a larger carbon dioxide footprint on CI/CD because more energy is used for compute-intensive tasks. One solution is to combine features, which leads to features that are not too big but also not too small. Another solution is to accumulate changes in

the pipeline for which resource-intensive tasks are executed less often; e.g., performing source code analysis only once a day. This leads to a separation of pipelines.

- *Application architecture*: One of the requirements in the previous chapter states that if the system consists of multiple microservices, each microservice should have its pipelines to guarantee the isolation of the microservices. This is similar to the statement in reference [6] that justifies that an application can have separate pipelines if parts of the application have different life cycles.
- *Operations pipelines*: Not part of the application pipeline, but one-off operations are typically realized using operations pipelines.

Delegation

An example of role separation concerns a quality assurance engineer who defines test cases, performs manual tests of the application, and automates the test cases as much as possible. Although integration of the automated tests in the pipeline is essential, some quality assurance engineers sometimes work in isolation, and the development of automated tests is separated from application development and pipeline development. At a certain moment, however, automated tests have to be integrated into the pipeline. This can be done using different techniques. One option is to add a *Perform test* stage to the main pipeline and implement the test tasks within that stage. Another option is to isolate the *Perform test* stage, implement the stage in a separate pipeline, and let the main pipeline invoke this *Perform test* pipeline. This means that the main pipeline does include a *Perform test* stage, but the execution of this stage is delegated

to another pipeline, maintained by the QA engineers. This gives them complete freedom of realizing automated tests, without much interference. Especially if the QA engineers are not part of the DevOps team that develops the application, this freedom is very welcome. This situation is visualized in Figure 4-76.

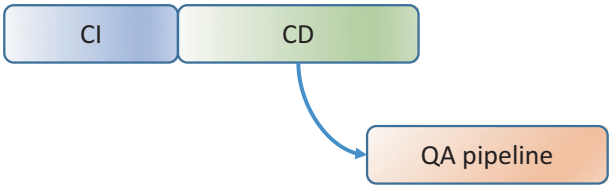


Figure 4-76. *Delegation of tests*

The separation of activities is visualized in Figure 4-77. The test pipeline is triggered from the main pipeline using a webhook trigger (triggers are explained in more detail in the next chapter).

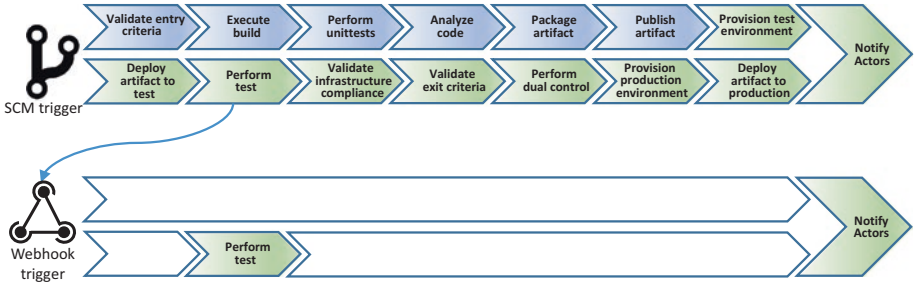


Figure 4-77. *Delegated QA pipeline*

Application Architecture

The architecture of the application has a large influence on the pipeline design. The pipeline design of a monolithic application consists of one artifact or multiple strongly coupled artifacts. This monolithic architecture differs from a microservice architecture. The pipeline design of an

"application" that consists of multiple microservices is a typical textbook example of a separation of concerns principle. The following are the characteristics of a microservice:

- Small in size
- Messaging enabled
- Bounded by contexts (organization around [business] capabilities instead of around technology)
- Autonomously developed
- Independently deployable, decentralized, and built and released with automated processes
- Can be implemented using different programming languages, databases, hardware, and software environment
- Decentralized data management with one datastore for each service
- Provides characteristics that are beneficial to scalability

This autonomy justifies separate pipelines for each microservice. So, if a team is responsible for three microservices, called A, B, and C, they need to develop three separate pipelines. See Figure 4-78.

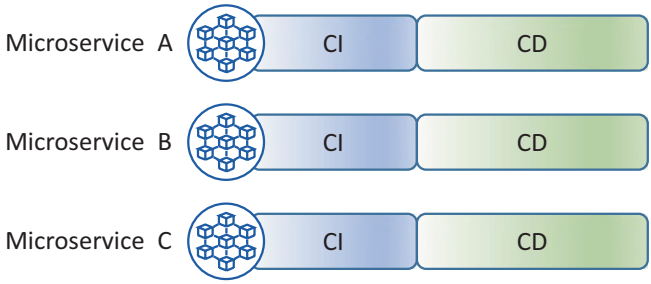


Figure 4-78. *Pipeline setup microservice architecture*

Orchestration

Sometimes it is needed that certain components are deployed in a particular order, or certain tasks need to be completed before a component can be deployed. This order of activities can be managed using an orchestrator pipeline. The orchestrator executes tasks and orchestrates the invocation of other pipelines. Consider a microservice architecture. In normal conditions, microservices run independently, so an orchestrator should not be needed at all. However, there could be a change in all microservices that justifies an order in deployment.[11] In Figure 4-79, the deployment order is managed by the orchestrator, first deploying microservice B, then microservice C, and finally microservice A. The orchestrator acts as an automated "runbook" to guarantee the order.

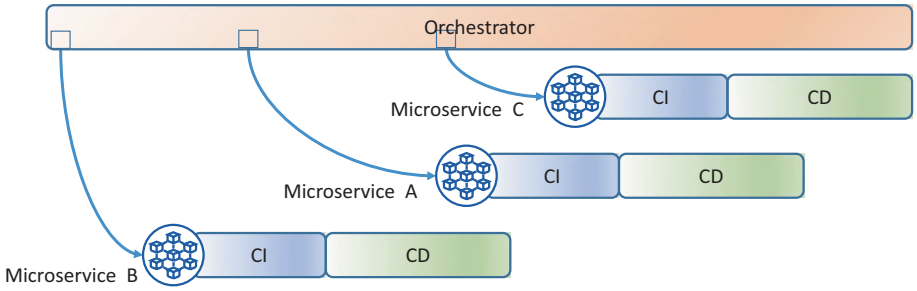


Figure 4-79. *Orchestrator*

The lifespan of the orchestrator varies, depending on the context. It may be a permanent pipeline in the pipeline landscape or a one-off pipeline that takes care of managing activities that are executed only once.

[11] Microservices are loosely coupled but not decoupled. If a new mandatory element is added to an event between two microservices, both microservices are impacted.

Event-Based CI/CD

All design strategies and considerations so far are based on a predefined workflow model. From a separation of concerns point of view, the stages of the workflow are divided over different pipelines. But what if we take this a level higher and consider an event-based CI/CD model? Similar to an application architecture in which a monolithic application is broken down into several microservices, it is also possible to do this for pipelines. The pipeline stages are developed as microservices, using an event-driven communication model. Each microservice consumes events and produces events. The events are specified according to a well-defined schema containing the metadata each microservice needs. External systems like source code management systems and issue trackers are hooked into the eventing framework and also produce and/or consume events.

The event-based CI/CD model of the Generic CI/CD Pipeline is transformed into Figure 4-80.

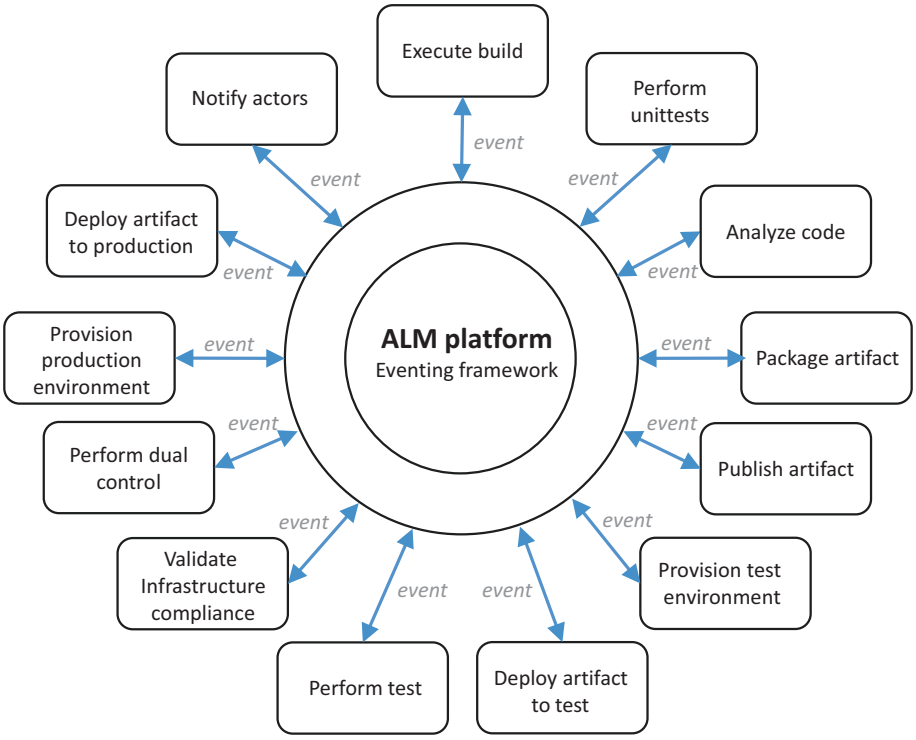


Figure 4-80. *Event-based CI/CD model*

Each microservice represents a pipeline stage. It is not part of a pipeline, but it is a self-contained piece of functionality that submits or listens to events, submitted by other microservices or events from external tools in the CI/CD ecosystem.

Note The *Validate entry criteria* and *Validate exit criteria* stages are gone, or at least not realized as separate microservices. These stages do not make any sense in the event-based model. This does not mean they are completely gone. Validating entry- and exit criteria tasks are now embedded in each microservice to guard the integrity of input and output data.

Each microservice publishes events and subscribes to certain topics. Each event consumed by the microservice is validated, based on the metadata the event carries. Based on certain rules, the microservices stage knows what to do and performs its actions.

- The *Execute build* microservice, for example, subscribes to a *git_push* topic. A *git_push* event triggers the *Execute build* microservice, which starts building the artifact.
- As soon as the *Execute build* microservice is successfully finished, it publishes an *artifact_built_success* event.
- Microservices *Perform unit tests*, *Analyze code*, and *Package artifact* are subscribed to the *artifact_built_success* topic and are triggered by the *artifact_built_success* event.
- Publishing an artifact is done only if all previous stages were successfully finished. The *Publish artifact* microservice subscribes to the *unit_test_success*, *analyze_code_success*, and *package_artifact_success* topics.
- If all three events on this topic are consumed by the *Publish artifact* microservice, the artifact is published. The *Publish artifact* microservice uses a Complex Event Processing (CEP) pattern to determine that the package can be published.
- The *Notify actors* microservice subscribes to any *_*success* and **_failed* topic and informs the actors in case such an event occurs.

This model has a few benefits over a pipeline model, listed here:

- Regular pipelines contain a mix of functionality and workflow, often closely integrated. In the case of an event-based CI/CD setup, there is no predefined workflow,[12] which means that if changes are made to a team's way of working, the event-based CI/CD setup is easily adaptable.
- A pipeline model still combines several stages into one pipeline. The development of these pipelines is difficult to perform in isolation. Even if developers, Ops engineers, and QA engineers are involved with pipeline development, they still have to cooperate closely and work on the same pipeline codebase. The event-based model decomposes the pipelines into individual microservices, operating fully independently. This provides the same benefits as application-based microservices, including autonomous development, deployment, and running instances in isolation.
- Autonomous development also implies that different roles can focus on the development of specific microservices without much interference.
- Parallelism is implicit. Multiple microservices subscribed to the same topic all start their execution as soon as an event on this topic is published.

[12] This is also a downside of an event-based model; if the workflow is not explicit anymore, one can lose track of the workflow as a whole.

Support for event-based CI/CD is limited to nonexistent in major ALM/integration platforms, but the idea is being embraced by some companies. Time will tell whether CI/CD migrates to the event-based model or whether the pipeline model remains the dominant approach.

Resource Constraints

Resource constraints come to light only when the pipeline is already developed and deployed. These resource constraints usually manifest themselves due to a lack of computing or storage resources. This results in a bad performance of the pipeline, or pipelines are put into a queue, waiting for an agent or compute node to become available. The simple answer to this problem is to add more hardware, but this is only one part of the story as we have seen. At some point, all options are stretched so far that other solutions have to be considered. Some of these solutions are ALM or integration platform related. Other options can be found in redesigning parts of the pipeline in such a way that their resource consumption is optimized. Here are some other considerations:

- *Revise the build strategy*: The build strategy was already explained earlier. Take a look at your build strategy again, and determine whether some things can be changed. Something as simple as pipeline caching improves performance a lot.
- *Priority clause*: The regular behavior of ALM/integration platforms regarding priority is that pipeline execution is first in, first out (FIFO). The problem is, if you deploy a "production fix," the pipeline execution joins the queue and is executed when all other pipelines in the queue are processed first. There is no distinction between a regular pipeline run and a production deployment. Wouldn't it be great if we could add a clause like in Listing 4-1 to our pipelines?

Listing 4-1. Priority Clause

```
priority:
  scope: global # Concerns the whole organization
  target: prod  # Deals with production; increased prio
  management-class: incident # Incident; more important
```

When using a priority clause like this, the particular pipeline queue is rearranged, and high-priority pipeline instances are moved to the front of the queue. Certain properties indicate how the queue is rearranged. Incidents in production have more priority than regular deployments to production. Regular deployments to production have more priority than regular deployments to test, etc. Unfortunately, few ALM/integration platforms offer prioritization of pipelines, and if they do, it is only rudimentary.

As an alternative to a priority clause, you can also define a pipeline setup with different execution environments (e.g., runners, executors, or agents). This way it becomes possible to define separate pipeline "lanes" in which pipelines of different categories run but don't interfere with each other.

- *Schedule pipelines*: Sometimes there are good reasons why a stage doesn't have to be executed multiple times per day. Analysis of source code can be done as part of the regular pipeline, but if multiple minor changes are applied to the codebase daily, the analysis of the source code often doesn't show much difference during that day. It makes sense to schedule source code analysis once a day in a quiet moment.
- *Limit continuous deployments*: Resource constraints can also be present in test environments. Even if the ALM/integration platform itself is capable of executing all pipelines fast enough, the test environment may

become a bottleneck. Separating the CI pipeline from the CD pipeline can help so the CD pipeline runs independently. The CI pipeline still runs after every code push, while the CD pipeline runs less often, and tests are executed less frequently.

- *Apply a resource lock*: In line with this is the use of a resource lock. If a test environment is still processing the tests of one pipeline instance, the next pipeline should not already be deploying another version of the application while the previous tests are not completed yet. To prevent this problem, a resource lock can be added to the pipeline. The resource lock prevents other pipelines from continuing their tasks until a given resource—e.g., a test environment—is ready and released back to the pool. One example of this is the Lockable Resources plugin in Jenkins. The downside of using resource locks is that it causes queuing.
- *Re-evaluate the execution of stages*: If the pipeline is started because a change has been pushed to a feature branch, is it really necessary to perform the *Analyze code* stage? Maybe the execution of this stage is not needed for a feature branch. If the team uses a more complex *workflow*, it may suffice that certain stages are executed only for specific branches. So, if there is a resource constraint, re-evaluate the pipeline stages for certain branches and decide whether they are required.
- *Parallelize stages and tasks*: Let's pick one case in which we look a bit closer at the possibilities of parallelized stages and/or tasks. If the codebase of the application is large, the *Analyze code* stage can become a compute-intensive stage that takes a long time to run. The

Generic CI/CD Pipeline has all stages ordered in sequence, which results in a pipeline in which *Execute build*, *Perform unit tests*, and *Analyze code* are executed after one another. Consider the case in which *Analyze code* itself contains three tasks: a SonarQube scan, a Fortify scan, and a Whispers scan. This results in the pipeline design shown in Figure 4-81.

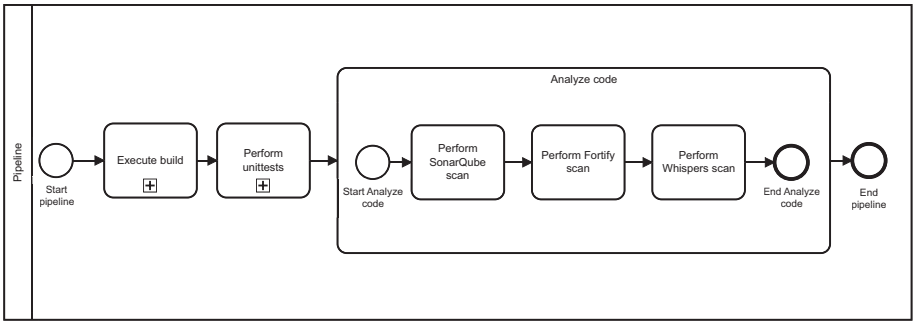


Figure 4-81. *BPMN, stages and tasks in sequential order*

Because the *Perform unit tests* stage depends on the artifact produced by the *Execute build* stage, both stages must be executed in sequence. The *Analyze code* stage, however, does not necessarily depend on the artifact, but on the source code in the repository.[13] Reordering the stages would result in a slight change in the design; see Figure 4-82.

[13] SonarQube requires an artifact, but this can still be a task detached from the creation of a regular build artifact.

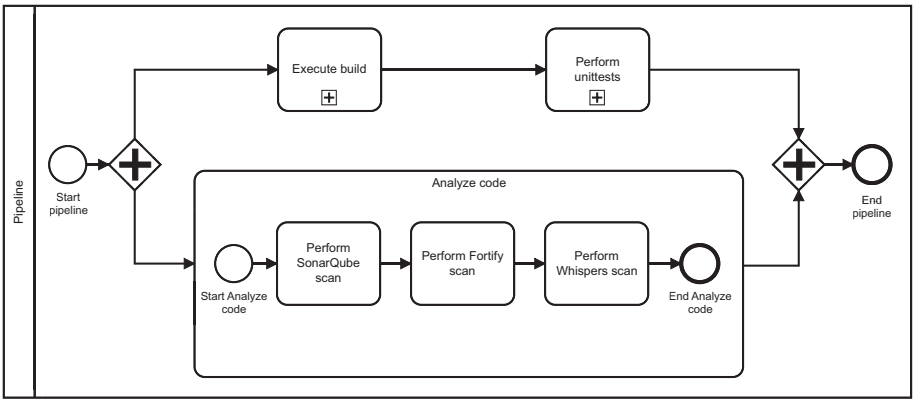


Figure 4-82. *BPMN, analyze code stage in parallel (with barrier)*

This design already reduces the overall processing time of the pipeline. Also, notice the use of the parallel gateway at the end of the *Perform unit tests* and *Analyze code* stages. In workflow modeling, this parallel gateway represents a "join." In multithreading, this is called a *barrier*. The barrier takes care that both *Perform unit tests* and *Analyze code* stages are completed before the pipeline continues (in this design example, the pipeline ends). This can be a requirement in case further testing should be performed; continue only in case both previous stages were completed successfully. Removing the barrier results in a pipeline design in which the *Analyze code* stage still executes in parallel, but the *Perform unit tests* stage (which isn't included in this model) doesn't wait for it to be completed, completely disregarding the result of the *Analyze code* stage. See Figure 4-83.

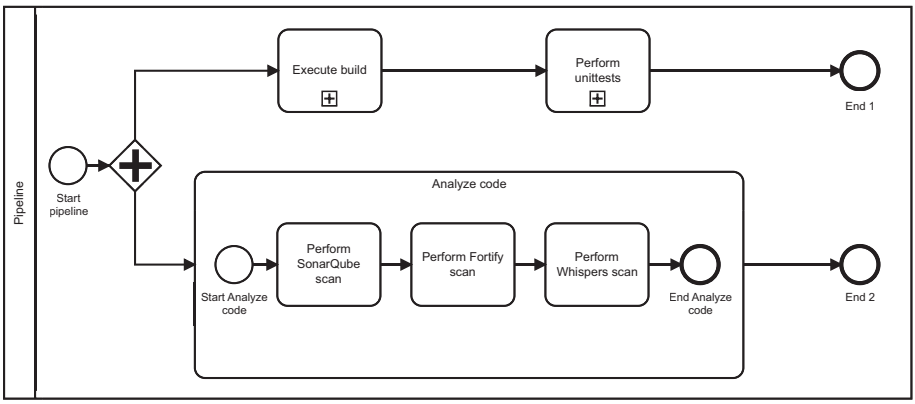


Figure 4-83. *BPMN, analyze code stage in parallel (without barrier)*

Taking a closer look at the three code analysis tasks reveals that also these tasks are independent. Applying further parallelization results in the design shown in Figure 4-84. The design makes use of a barrier (parallel gateway) at the end of the *Perform unit tests* and *Analyze code* stages, but also the individual tasks of the *Analyze code* stage end with a barrier; the *Analyze code* stage ends only if all three tasks are completed.

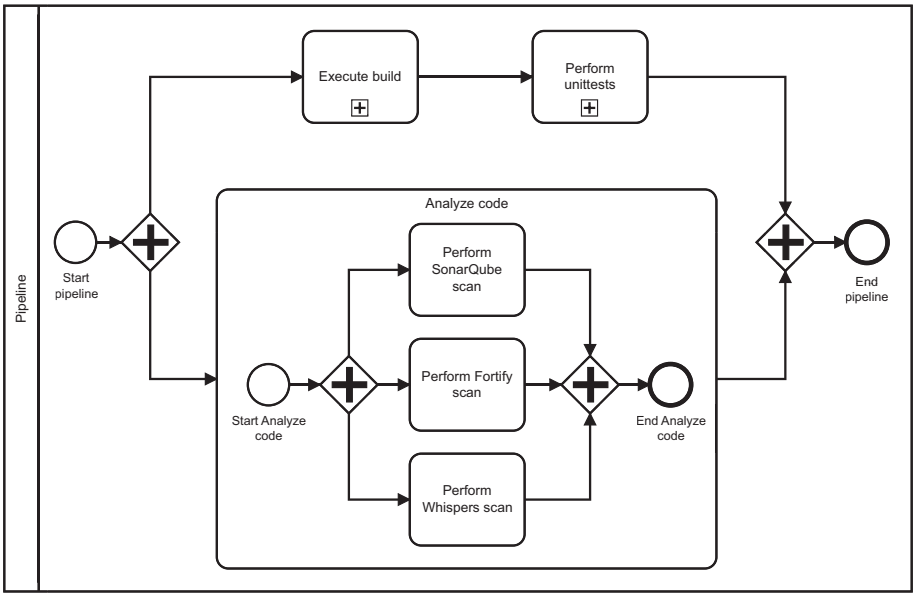


Figure 4-84. *BPMN, analyzing code tasks in parallel (with barrier)*

If the infrastructure can process everything in parallel, the overall processing time would decrease even further. Chapter 6 shows that this theory results in a better performance of the pipeline.

This setup works only if there are enough resources available to process everything in parallel. If resources are not sufficient, the whole *Analyze code* stage can be detached from the main pipeline and wrapped in a pipeline that runs only once a day, represented in the BPMN diagram shown in Figure 4-85.

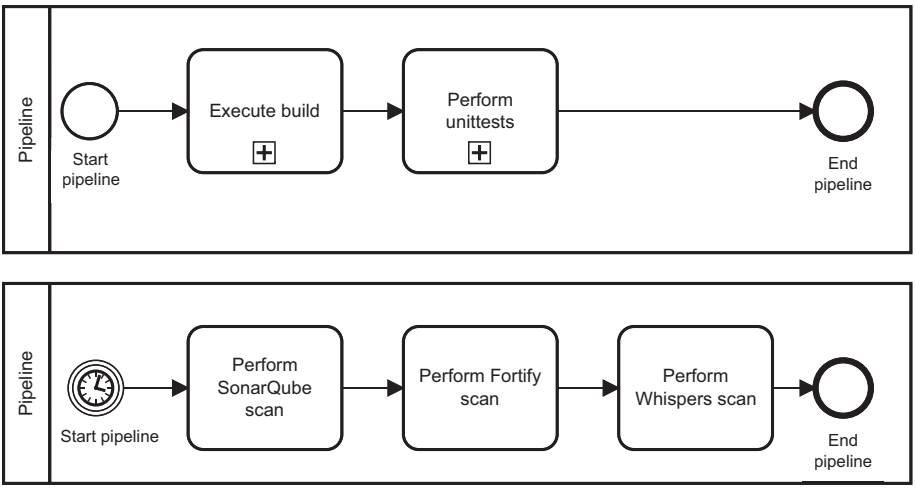


Figure 4-85. *BPMN, scheduled Analyze code pipeline*

Commercial Off the Shelf

CI/CD is always discussed in the context of developing software in-house. Vendor packages—from a consumer point of view—are left out of scope, which makes perfect sense. Vendor packages, or the so-called commercial off-the-shelf (COTS) applications, are already developed and tested, so it is just a matter of downloading, installing, and using them, right?

CI/CD from a vendor point of view also puts constraints on the process. The client chooses the life cycle of installing the software. This means that deployment to production has a different meaning for vendors of software packages. There is a clean separation between delivering production-ready software and delivering the software to clients.

Clients of COTS software have a hard time understanding how the installation of vendor packages benefits from the CI/CD concepts. Often, installation of these packages is a manual or semi-automated process, prone to errors and with an increased risk of fraudulent handling. While the activities of installing vendor packages differ from developing software in-house, there are great benefits to gain in formalizing these activities

using a pipeline. When using an (automated) pipeline, the steps are performed in a controlled way, which makes it possible to have an audit trail of the process. Let's look at the steps involved in the case of the installation of a vendor package. The context is a closed COTS solution from a consumer point of view, with only binaries supplied.

The following are the stages included in the COTS pipeline:

- *Validate entry criteria*
- *Download package*
- *Validate integrity*
- *Publish package (internal)*
- *Provision test environment*
- *Install and configure in test*
- *Test/validate the application*
- *Approve production installation*
- *Provision production environment*
- *Install and configure in production*

Here they are in more detail:

- *Validate entry criteria*: The first step is to verify the version that needs to be installed. Also, make sure to choose the binary for the appropriate operating system. Check and authenticate the vendors' endpoint/URL from where the package is downloaded, especially if it is one of the first times the package is downloaded.
- *Download package*: Package solutions are often retrieved from the vendor using a portal to which a user logs in. The package (application) is downloaded and stored in a temporary location. Some vendors provide

an API that can be used to download the package. The use of an API is preferred over the use of a portal. Make sure the metadata is stored; credentials used to log in on the vendor's system and the date and time the package was downloaded. Also, store the downloaded package in a secure location within the boundaries of your on-premises datacenter (or cloud account).

- *Validate integrity and vulnerabilities*: After the package is downloaded and stored in an intermediate storage location, it is checked for integrity. The integrity of the downloaded package is validated by verifying the hash or a digital signature. Validating a digital signature guarantees that nobody has tampered with the package. Hash validation is, from a security point of view, a weak mechanism to validate integrity. In addition, the package must be scanned for viruses and malware.
- It cannot be assumed that vendor software does not contain vulnerabilities. That is why this type of software must also be scanned for vulnerabilities (if possible), such as the use of third-party libraries and plugins.
- *Publish package (internal)*: The artifact downloaded from the vendor must be stored in an immutable binary repository, including any additional metadata, such as release notes from the vendor. Storing the package in a repository guarantees that its integrity remains. The intermediate location to store the downloaded package and the immutable repository can be the same location by the way.

- *Provision test environment*: Depending on the requirements, it makes sense to provision a sandbox environment that does not allow outbound communication and prevents malicious software from scanning the network and/or setting up a communication session with a server outside the organization. This test environment is used as a sandbox environment to install the downloaded package. Additional security measures take care that the software does not become rogue and perform unintended actions.
- *Install and configure in test*: This step involves the installation and configuration of the package in the test environment. This can be a manual, a semi-automated, or a completely automated task. The complexity of this stage varies. If the current release is too far behind, a complete migration needs to be implemented. If the difference between the old and the new version is small, the risk to update with the new version is low. A security test in a sandbox environment may be part of the tests involved.
- *Test/validate the application*: Although the application is already fully tested by the vendor, some form of a smoke test is still needed to determine whether the configuration is done properly and to test whether the integration with surrounding systems (still) works. More extensive testing is needed if the package is integrated with another system such as an IAM system or a customer relationship management (CRM) system. Also, performance testing may be included in this stage.

- *Approve production installation*: If the application behaves as expected, it is approved. Also, this can and should be implemented using a dual control step.
- *Provision production environment*: The production environment—if not already available—is created.
- *Install and configure in production*: This step involves the installation and configuration of the package in the production environment. This can be a manual, a semi-automated, or a completely automated task.

This leads to a pipeline similar to the one in Figure 4-86 and Figure 4-87.

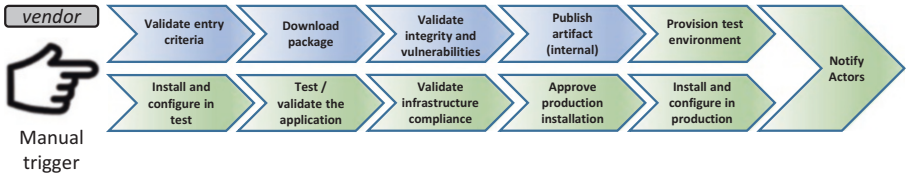


Figure 4-86. *Commercial off-the-shelf pipeline*

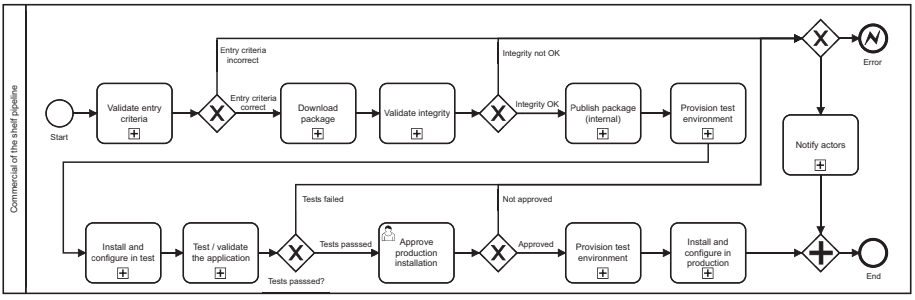


Figure 4-87. *BPMN, commercial off-the-shelf pipeline*

Note Automating these stages as much as possible provides great benefits. It speeds up the process, is more reliable, and is more secure. But even if the complete process is done manually, a pipeline with these stages is still useful. The stages in this manual process are discrete and can be completed with a sign-off task. All pipeline instances (runs) and meta-information (such as sign-offs, credentials, and timestamps) are stored in the ALM/integration platform, which makes the process fully transparent and auditable.

Summary

You learned about the following topics in this chapter:

- Modeling a pipeline flow in BPMN 2.0
- Drafting a context diagram and using it as a means to communicate with the team
- Using the Generic CI/CD Pipeline as starting point for your pipeline design
- The effect of certain strategies (branching, build, test, release, and deployment strategy) on the pipeline design, how the flow changes if certain choices are made, and why, when, and how to split a pipeline in other independent pipelines
- How to apply these strategies to your situation
- How other factors, such as separation of concerns and resource constraints, affect the pipeline design
- How commercial off-the-shelf application benefit from a pipeline implementation, because it formalizes the stages and tasks

CHAPTER 5

Pipeline Development

This chapter covers the following:

- The different types of pipeline specifications
- The features used in the different ALM/integration platforms, along with some code snippets to show the pipeline code benefits if these features are offered as code constructs
- The security issues when dealing with external libraries as well as solutions on how to mitigate them
- How the target environment properties can be stored and used in the pipeline
- Secrets management and how to mitigate security risks concerning secrets used in pipelines
- Feature management and the different ways to apply it
- The levels in the organization in which CI/CD-related development occurs and the different ways DevOps teams develop their pipelines
- Practical tips for sustainable pipeline development

A chapter about developing pipelines that still tries to preserve the abstract character of this book almost seems an impossible assignment. The platform landscape is wide with a plethora of tools to choose from,

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_5

each with its characteristics and technical solutions. Still, various generic topics can be emphasized, even if the implementation is different. This chapter discusses some of these topics and examples that deal with pipeline development.

Pipeline Specification

A pipeline specification covers the translation of the logical pipeline design into a technical definition. This results in one or more files containing pipeline code executed on an ALM/integration platform.

Multibranch, Multistage Pipeline

The features added to the various ALM/integration platforms have increased over time, and these platforms have become more mature. In the past, pipelines were simple, but nowadays it is possible to develop pipelines with a more complex flow. Out-of-the-box functionality, plugins, and marketplace solutions enable feature-rich pipelines yet avoid plumbing code. Activities are grouped into discrete stages, jobs, and tasks, making it possible to parallelize work, reduce execution time, and allow faster feedback to the developer.

The days that a pipeline could be used in combination with code from only one SCM branch are over. Pipelines can be triggered if a change in any branch of the repository has been made. The pipeline decides what to do, depending on the branch, and certain conditions. These multibranch, multistage pipelines are very powerful and make it possible to develop complex automation processes. This chapter shows some features and possibilities of modern pipeline development and specification.

Pipeline specification cannot be generalized, because different tools use different language constructions and have different features, but in general, there are three ways to create a pipeline.

- Using a user interface
- Using a scripted pipeline
- Using a declarative pipeline

Let's go through these options.

User Interface–Based Pipelines

Most ALM/integration platforms such as Jenkins, Bamboo, and Azure DevOps include user interfaces to create pipelines. This provides a graphical view of a pipeline but also offers a fast and more intuitive way to create pipelines. Using a user interface also has downsides. Some user interfaces are cluttered, and certain options are well-hidden in the caverns of the user interface. In addition, user interface–based pipelines usually do not support version control of the pipelines. Of course, in some cases, it is possible to export a pipeline as a file and manage it in an SCM, but this is a rather cumbersome workflow. In general, use a graphical user interface only in the case of a simple pipeline that can be re-created easily, or use it to learn how a pipeline is constructed. In all other cases, use scripted or declarative pipelines.

Figure 5-1 illustrates a Jenkins freestyle project. It shows the user interface used to create a pipeline. It allows adding multiple build steps to a pipeline. However, it is also limited in its capabilities.

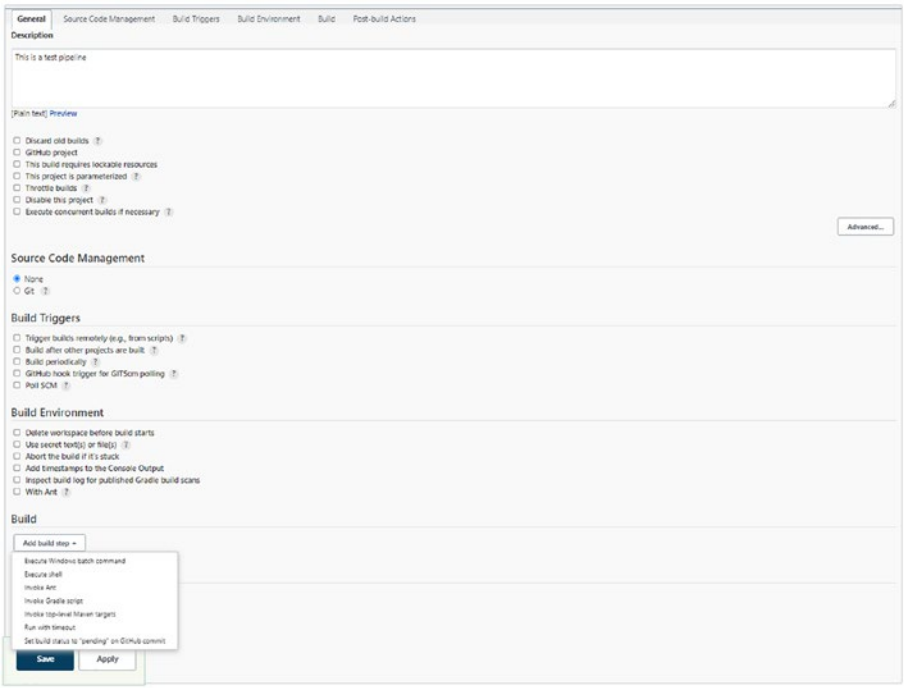


Figure 5-1. *Jenkins freestyle project*

Scripted Pipelines

A scripted pipeline is either a file containing a scripting language or a domain-specific language (DSL) language, but it can also consist of a complete project, supported by a general-purpose programming language. An example of a scripted pipeline is the Groovy pipeline used in Jenkins. Atlassian's Bamboo has the option to develop a pipeline based on a complete Java project (Bamboo Java Specs).

Besides the benefit that scripted pipelines are just files, which can be put under version control, scripted pipelines are also extremely versatile. You have full control of the flow and the implementation of the stages

and tasks. However, this can also become a pitfall. If not taken care of, the pipeline code becomes unreadable. Listing 5-1 shows a simple structure of a scripted Jenkins pipeline.

Listing 5-1. Jenkins Script

```
node {
    stage(' Stage 1') {
        //
    }
    stage(' Stage 2') {
        //
    }
    stage(' Stage 3') {
        //
    }
}
```

Declarative Pipelines

Declarative pipelines are similar to scripted pipelines, but they have a more restricted syntax that preserves the pipeline structure and prevents the code from becoming bloated and unreadable. Declarative pipelines intend to be better structured, which makes reading and writing the pipeline code easier. This does not mean you cannot do the things you can do with scripted pipelines. It is common to add scripting to a declarative pipeline, but because of the strict syntax, the scripting has a distinctive place in the pipeline structure. The trend seems to be shifting toward the use of declarative pipelines, and especially YAML-based pipelines dominate the pipeline landscape.

Consider a team using a feature branch workflow. Their integration platform of choice is Jenkins. The Jenkins pipeline is stored in a source code management repository as a file called the Jenkinsfile.

The basic structure of the *Generic CI/CD Pipeline* in declarative Jenkins code looks like Listing 5-2.

Listing 5-2. The Generic CI/CD Pipeline in Jenkins Declarative Code

```
pipeline {
    agent any

    stages {
        stage('Validate entry criteria') {
            steps {
                echo 'Stage: Validate entry criteria'
            }
        }
        stage('Execute build') {
            steps {
                echo 'Stage: Execute build'
            }
        }
        stage('Perform unit tests') {
            steps {
                echo 'Stage: Perform unit tests'
            }
        }
        stage('Analyze code') {
            when {
                branch "main"
```

```
    }
    steps {
        echo 'Stage: Analyze code'
    }
}
stage('Package artifact') {
    steps {
        echo 'Stage: Package artifact'
    }
}
stage('Publish artifact') {
    steps {
        echo 'Stage: Publish artifact'
    }
}
stage('Provision test environment') {
    when {
        branch "main"
    }
    steps {
        echo 'Stage: Provision test environment'
    }
}
stage('Deploy artifact to test') {
    when {
        branch "main"
    }
    steps {
        echo 'Stage: Deploy artifact to test'
    }
}
```

```
        stage('Perform test') {
            when {
                branch "main"
            }
            steps {
                echo 'Stage: Perform test'
            }
        }
        stage('Validate infrastructure compliance') {
            when {
                branch "main"
            }
            steps {
                echo 'Stage: Validate infrastructure compliance'
            }
        }
        stage('Validate exit criteria') {
            when {
                branch "main"
            }
            steps {
                echo 'Stage: Validate exit criteria'
            }
        }
        stage('Perform dual control') {
            when {
                branch "main"
            }
            steps {
                echo 'Stage: Perform dual control'
            }
```

```
        }
        stage('Provision production infrastructure') {
            when {
                branch "main"
            }
            steps {
               echo 'Stage: Provision production infrastructure'
            }
        }
        stage('Deploy artifact to production') {
            when {
                branch "main"
            }
            steps {
                echo 'Stage: Deploy artifact to production'
            }
        }
    }

    // Stage: Notify actors
    post {
        success {
            echo 'Stage: Notify actors - success'
        }
        failure {
            echo 'Stage: Notify actors - failure'
        }
    }
}
```

This *Jenkinsfile* contains only the skeleton of the feature branch workflow. Notice that all stages are executed if the branch is main. In the case of a feature branch, only a subset of the stages is executed.

Assuming that the *Jenkinsfile* is included in the same repository as the application, the workflow of the team has to be adopted when changes are applied to the *Jenkinsfile*. Changes to the *Jenkinsfile* are done in a feature branch and, when finished, merged with the mainline. This makes testing of the *Jenkinsfile* a bit problematic because only a subset of the flow can be tested, namely, the stages associated with the feature branch. Testing the stages associated with the main branch is not straightforward, and also destructive actions in the pipeline must be mitigated. To solve this, we need some way to properly test pipelines. The next chapter shines some light on testing pipelines.

Constructs

One of the issues with pipelines is that complex actions sometimes require a lot of plumbing code. Declarative YAML-based pipelines are also not very versatile, because YAML is not a real programming language. Complex setups such as canary deployment or building various versions for different target environments blow up the pipeline declaration, are hard to read, and are difficult to maintain unless there are features in the platform supporting this complexity.

A *construct* is a generic name for pipeline features that reduce complexity. Constructs are out-of-the-box features solving problems not easy to solve otherwise. This paragraph is devoted to some of the (common) constructs found on various platforms. The examples are not "taken" from only one platform, but from various ones. Not all platforms support all constructs. The examples are to show only what is possible.

Triggers

There are several ways to start a pipeline, depending on the context. Starting a pipeline is based on triggers, and most ALM/integration platforms support various kinds of triggers. These are the most common ones:

- *SCM trigger*: Most common is the SCM trigger that starts a pipeline after code is committed and pushed to a source code management repository. The pipeline builds the artifact based on the branch in which the code was committed. In addition to code pushes, other SCM events may lead to triggering a pipeline. One example is an event submitted after a pull request has been approved. SCM triggers can be implemented using webhooks or as an integrated feature of an ALM/integration platform.

Tip If you plan to incorporate the pipeline file into the same source code repository as the application, remember that if you use an SCM trigger, the pipeline by default also runs after you changed the pipeline code itself, which potentially could lead to the deployment of the application to production (or at least to a test environment). It is better to move the pipeline code to a separate directory and exclude this directory from the trigger; this option is provided by several platforms. An alternative is to exclude the pipeline file(s) based on the filename or extension if the platform supports this feature.

- *Webhook*: A webhook refers to an API callback that starts a pipeline. The API is part of the ALM/integration platform that can be used by external systems to trigger

a pipeline. It receives an HTTP request, containing meta-information. A big advantage of webhooks is that the calling system does not have to be an integrated subsystem of an ALM platform. It can be a stand-alone tool triggering the pipeline. A nice example is the support of webhooks in GitHub. The webhook can be enabled not only when code is pushed but also for other types of events. By enabling the webhook in GitHub and configuring the pipeline endpoint, the endpoint is invoked every time a certain event in GitHub is published. The pipeline endpoint can be an external integration server like Jenkins. Webhooks are usually not defined in the pipeline declaration.

Note Beware of a potential security vulnerability when using webhooks. In the case of an SCM trigger or a manual trigger, the user is known, so a dual control in the pipeline can exclude this user from approving their own change. In the case of a webhook, the credentials with which a pipeline is triggered are often different (e.g., a nonpersonal account). So, if someone can invoke the webhook, they may also be able to approve the pipeline in the dual control (the credentials of the webhook and the credentials of the person performing the dual control differ).

- *Schedule*: Schedules are a way to define at which moments the pipeline must start. This can be once a day, once a month, or every minute. The most versatile way to specify a schedule is using a cron expression. Listing 5-3 shows an example.

Listing 5-3. CircleCI, Scheduled Trigger; Every Working Day at 10 p.m.

```
workflows:
  at_ten:
    triggers:
    - schedule:
      cron: "0 22 * * 1-5"
```

In Listing 5-3, a trigger is configured, which starts the pipeline every working day at 10 p.m.

- *Pipeline completed*: There are several ways a pipeline can be started by another pipeline. A pipeline can be triggered using a webhook, in which the invocation of this webhook is explicitly added to the calling pipeline. This can be done by adding a `curl` command (on Linux) to the pipeline definition, but this is not a very clever solution. If the endpoint of the other pipeline changes, the calling pipeline must be changed also. A better way is to use a pipeline complete construct in the pipeline that needs to be triggered. In this pipeline, it is defined to which other pipeline(s) it "listens." The pipeline completed construct is a typical example of an Observer pattern implementation. In the example shown in Listing 5-4, a pipeline is started as soon as another pipeline with the name `pipeline-that-triggers-me` is completed.

Listing 5-4. Azure DevOps, Pipeline Triggered by Another Pipeline

```
resources:
  pipelines:
  - pipeline: logical-name-of-this-pipeline
    source: pipeline-that-triggers-me
    trigger: true
```

This pipeline, with the name `logical-name-of-this-pipeline`, is started after the `pipeline-that-triggers-me` is completed.

- *Manual*: A pipeline can always be started manually, of course. Usually, no specific declaration needs to be added to the pipeline to make this possible.

Execution Environment

Modern platforms provide the option to specify in which environment a pipeline is supposed to run. The various platforms use concepts like "slave" nodes, runners, executors, or agents, whether grouped into a pool of servers or containers. In essence, the execution environment is the environment in which a pipeline runs. This can be in the form of a Linux or Windows server, but it is also possible to execute a pipeline in a Docker container running on a (Kubernetes) cluster. These environments are preconfigured and registered to the ALM/integration platform. These environments also consist of preconfigured tools. If you want to build an artifact using Java or Python, the environment must have pre-installed Java JDK and Python.

In addition to running the whole pipeline in one specific environment, it is also possible to decompose the pipeline and have each part of the pipeline run independently. The pipeline is decomposed, often as so-called jobs. Each job is executed in a specific environment. Jobs of one

pipeline may run in the same environment, but jobs may also run in separate environments. This also means that in these situations there is no shared memory and passing information between jobs is not always trivial. Listing 5-5 and Listing 5-6 show some examples.

Listing 5-5. CircleCI, Job Executed in a Docker Container

```
jobs:
  build:
    docker:
      - image: cimg/openjdk:17.0.3
```

Listing 5-6. Azure DevOps, Job Executed on a Self-Hosted Server

```
jobs:
- job: build
  pool: myServerPool
```

Listing 5-5 defines a Docker container with a base image containing the OpenJDK. This becomes the runtime environment of the pipeline. Listing 5-6 defines a self-hosted server pool—myServerPool—consisting of servers on which the pipeline runs. The pool may consist of one or more servers with a certain operating system and pre-installed tools.

These constructs are simple yet powerful. With only a few lines of code, it is possible to declare where a pipeline or even individual jobs are executed, and the platform takes care of it.

Connections

Pipelines often connect to external systems with a specific endpoint, a certain protocol, and security credentials. Using `curl` in the pipeline to connect to an external Nexus IQ server may work, but this does bloat the pipeline code. A more elegant way is to make use of *connectors* or *service*

connections. Various platforms name them differently, but in essence, these connectors are endpoint specifications defined in a special—and secured—connection store. This endpoint is referred to in the pipeline by its logical name, which results in a cleaner and more secure pipeline declaration, and prevents you from having to store a username and password in an SCM. In addition, some platforms support options to set up dual control for creating service connections.

Listing 5-7. Azure DevOps, Nexus IQ Service Connection

```
- task: NexusIqPipelineTask@1
  displayName: 'Nexus IQ policy evaluation'
  inputs:
    nexusIqService: 'ServiceConnectionNexusIQ'
    applicationId: myApp
    stage: 'AnalyzeCode'
```

Listing 5-7 refers to the ServiceConnectionNexusIQ service connection as the logical endpoint of NexusIQ. This endpoint is specified outside the pipeline declaration, as shown in Figure 5-2.

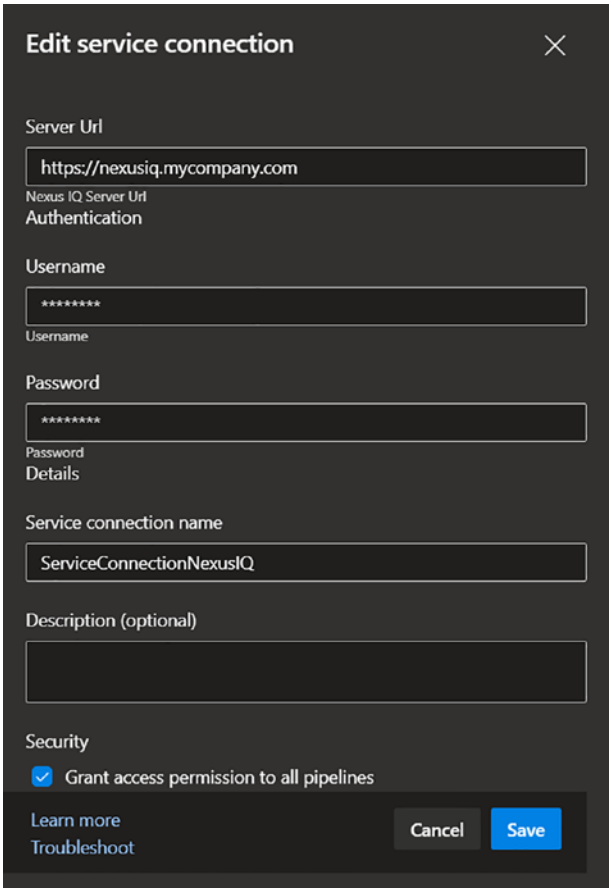


Figure 5-2. *Azure DevOps, service connection of NexusIQ*

Variables

Variables in pipelines are similar to variables in a programming language. Variables can be defined in a pipeline, but certain platforms also provide the option to define variables outside the pipeline specification, sometimes grouped with a logical name. Special care needs to be taken concerning variable scope. As explained earlier, parts of the pipeline—stages or jobs—can be executed on different runtime environments, which makes sharing variables more troublesome, or in some cases even impossible.

A special case of variables concerns conditional variables. Conditional variables are handy to assign a value to a variable, given a certain condition. For example, an HTTP endpoint of a test environment differs from the HTTP endpoint of a production environment. The `endpoint` variable in Listing 5-11 depends on the `target` variable.

Listing 5-11. Azure DevOps, Conditional Variable

```
variables:
  - name: endpoint
    ${{ if eq( parameters['target'], 'test') }}:
      value: 'https://mycompany.test.com'
    ${{ if eq( parameters['target'], 'production') }}:
      value: 'https://mycompany.com'
```

Conditions

Conditions in pipelines are indispensable. Conditions in scripted pipelines are implemented using an `if/then/else` construction. Conditions in declarative pipelines often have a different structure and use keywords like `if`, `when`, or `condition`, depending on the platform. Some examples of conditions on different platforms are shown in Listing 5-8, Listing 5-9, and Listing 5-10.

Listing 5-8. GitLab, `if` Example

```
job:
  script: echo " Run Analyze code in case of the main branch"
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

Listing 5-9. Jenkins, `when` Example

```
stage('Analyze code') {
    when {
        branch "main"
    }
    steps {
        echo 'Run Analyze code in case of the main branch'
    }
}
```

Listing 5-10. Azure DevOps, `condition` Example

```
- stage: Analyze_code_stage
  displayName: 'Analyze code'
  condition: eq(variables['Build.SourceBranchName'], 'main')
  jobs:
  - job: Analyze_code_job
    steps:
    - script: echo 'Run Analyze code in case of the
main branch'
```

Caching

Caching decreases the time to build an artifact. Different platforms have implemented caching in different ways. In one of the researched platforms (CircleCI), it is implemented as an integrated construct in the pipeline declaration and is accessed by using the `save_cache` and `restore_cache` keywords, while in other platforms, caching is added as a marketplace solution that performs the save and restore actions.

When using the caching feature, it becomes possible to store external libraries or even compiled code to a "cache store" and use this cache in subsequent pipeline runs. It is best to explain this using Figure 5-3.

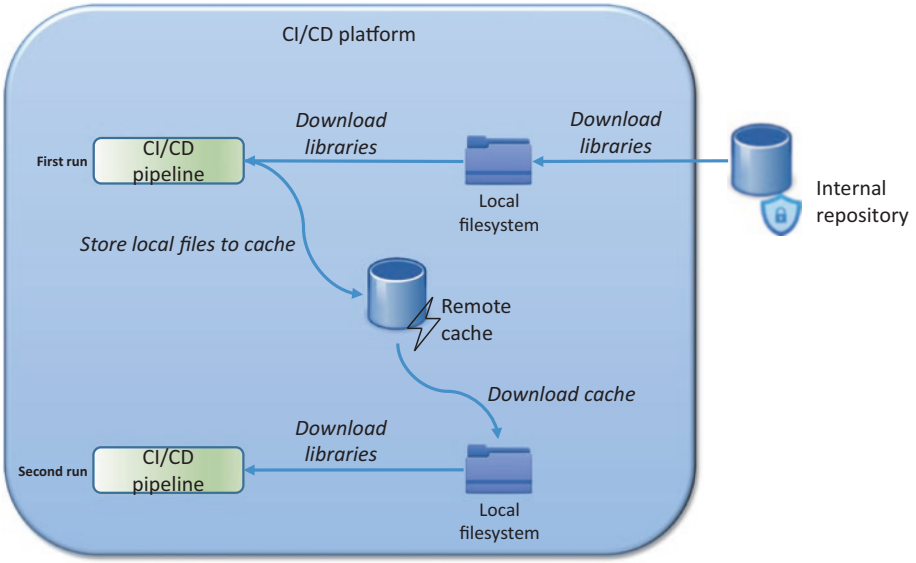


Figure 5-3. *Pipeline caching*

In the first pipeline run, libraries are downloaded from a repository as part of an artifact build task. These files are locally stored, so the pipeline can use them. If the pipeline is finished, the files are stored as a persistent cache (remote cache) for later use. In subsequent pipeline runs, the libraries are not retrieved anymore from the repository, but instead, the persistent cache is downloaded. Libraries are not downloaded from the Internet or a repository anymore in the artifact build task, but the files of the cache are used instead. This is much faster.

A common pattern in most platforms is to store the cache using a key. This can be a fixed key, like `myCache`, but often, caches are immutable, meaning they cannot be updated anymore after creation. A smart solution to tackle this problem is to hash specific files that declare the libraries and use the hash as part of the key. If one of these files changes because a different library version is defined, the key changes. This results in building up a new cache.

Take a Maven project. The project is configured using a `pom.xml` file (or multiple `pom.xml` files). Part of the key with which the cache is stored contains the hash of this `pom.xml`; or in the case of multiple `pom.xml` files, all files are hashed, and a new hash is created from the concatenated hashes. Listing 5-12 contains an example of the definition of an immutable cache.

Listing 5-12. Azure DevOps: Immutable Cache Definition

```
- task: Cache@2
  inputs:
    key: 'maven | "$(Agent.OS)" | **/pom.xml'
    restoreKeys: |
      maven | "$(Agent.OS)"
      maven
    path: $(MAVEN_CACHE_FOLDER) # is ./.m2/repository
  displayName: cache_maven_local_repo
```

The trick is to assemble a cache key, using the Maven prefix, the operating system, and all `pom.xml` files. The `**/.pom` pattern is used to calculate the hash of all the `pom.xml` files. As soon as one of the `pom.xml` files changes, the hash changes, and a new cache is saved and restored.

Listing 5-13. Azure DevOps, Log Determining the Cache Key

```
Resolving key:
- maven [string]
- "Linux" [string]
- **/pom.xml [file pattern; matches: 3]
- s/pom.xml -->
7CC04B8124B461613E167AA0D15E62306BDF553750988B6BF21355
E641B163DE
```

```
- s/app-cdk/pom.xml --> 73B0183B69BB3454081CBB6F2CE08176AAD82
D6CCB586ECE6368D617B632FD56
- s/s3-lambda/pom.xml --> 59D32A57C7138664E36F1C56CF319510B2
EC10A438ACB33059AA8DC95E3C0490
Resolved to: maven|"Linux"|L+f1r46o5J7Rhd43eGymkldHfa
5BAH5UHbZevoWBSco=
```

Note It is important to realize that the optimization step to retrieve something from a cache instead of the source should not compromise security. Platforms should take care that caches are scoped to specific pipelines and that the integrity of a cache is guaranteed. For the latter, adding a digital signature to a cache and verifying it when used would be the best solution. This does not seem to be a (transparent) feature in the major platforms, but it is possible to implement it in the pipeline yourself.

Matrix

A matrix is used to declare an action using all permutations of variables declared in the matrix. The matrix implements a *fan-out* pattern and can be used for the implementation of a cross-platform build strategy. Using a matrix, it becomes possible to define a build for multiple language versions and multiple target environments. Listing 5-14 shows an example of a matrix declaration.

Listing 5-14. GitHub Actions, Matrix Strategy Used in a Build Job

```
jobs:
  build:
    runs-on: ${{ matrix.os }}
```

```
    strategy:
      matrix:
        python-version: [3.7, 3.8]
        os: [ubuntu-latest, macOS-latest, windows-latest]
```

In Listing 5-14, six jobs are instantiated in which an artifact is built for two Python versions and three operating systems. The syntax of this declaration is elegant and simple and prevents the same code from being repeated six times in one pipeline declaration.

A matrix can be used for more than only building artifacts. It can also be used to test multiple versions of an artifact in parallel.

Deployment Strategy

A deployment strategy can become complex. There are various solutions to solve this problem. A common—and recommended—solution is to use a deployment tool with built-in deployment strategies. Examples are AWS CodeDeploy, which supports canary deployments, and Cloud Foundry CLI with the blue-green deployment plugin. Using specific deployment tooling has a lot of benefits, but sometimes it is not possible to use a tool. There can be a technical or financial constraint that "forces" teams to implement the deployment strategy in the pipeline itself.

Fortunately, some platforms have features that help implement deployment strategies in the pipeline. One of these features is the canary deployment construct shown in Listing 5-15.

Listing 5-15. Azure DevOps, Canary Deployment Strategy

```
jobs:
- deployment:
  environment: production
  pool:
    name: myAgentPool
```

```
  strategy:
    canary:
      increments: [10]
      preDeploy:
        steps:
        - script: "Performing initialization"
      deploy:
        steps:
        - script: echo "Deploying…"
      routeTraffic:
        steps:
        - script: echo "Route traffic to updated version"
      on:
        failure:
          steps:
          - script: echo "Deployment failed"
        success:
          steps:
          - script: echo "Deployment succeeded"
```

The deployment deploys in increments of 10 percent until it reaches 100 percent. During each increment, the traffic is routed to the new version until all traffic is directed to the new version and the deployment is completed. If the deployment fails, the deployment must be rolled back. This construct helps in structuring the pipeline declaration. Unfortunately, the actual implementation must still be coded.

Auto-cancel

If a pipeline contains a task to sign off a manual test result and this pipeline is executed multiple times, multiple orphaned pipeline instances pile up and wait for a manual sign-off. The previous chapter proposes various solutions. One of them is to use the "auto-cancel" option. With an auto-

cancel construct, all already running instances of the same pipeline are canceled if a new pipeline instance is started. The new instance always includes the latest code changes. This means there are no dangling pipelines anymore.

Listing 5-16. Semaphore, Auto-cancel

```
auto_cancel:
  running:
    when: "true"
```

There are similar constructs that almost do the same, but not quite. Azure DevOps has a "batch" feature. Enabling the "batch" option does not start any new instance of the pipeline if there is still a running instance.

On Success/Failure

Just as in regular programming languages, there is a need to add a `try/catch/finally` construct in a pipeline. They come in various flavors. Sometimes—in scripted pipelines—they are just implemented as `try/catch/finally` blocks. In declarative pipelines, you see implementations like a `post` section, which includes blocks that can be executed conditionally.

Listing 5-17. Jenkins, Post Success/Failure

```
post {
    success {
        echo 'Stage: Notify actors - success'
    }
    failure {
        echo 'Stage: Notify actors - failure'
    }
}
```

Listing 5-18. Azure DevOps, On Success/Failure

```
on:
  success:
  - script: "Notify actors - success"
  failure:
  - script: "Notify actors - failure"
```

It is important that these constructs can be used on different levels within the pipeline. Using them within a stage deals with stage-scoped issues. Using them on a pipeline level means that the scope applies to the whole pipeline.

Fail Fast

One of the key elements in CI/CD is to fail fast and return immediate feedback. This concept is implemented differently on each platform, and there is no generic construct that has been adopted by multiple platforms. A *fail fast* means that if a stage, job, or task fails, the whole pipeline stops immediately. The example in Listing 5-19 stops all jobs in the pipeline in the case of an error.

Listing 5-19. Semaphore, Fail Fast

```
fail_fast:
  stop:
    when: "true"
```

Priority

It was already mentioned earlier, but prioritizing pipelines is a must-have feature. In addition, it should be possible to define this prioritization on different levels. A pipeline run solving a production incident should have priority over previous nonurgent pipeline runs. In addition, priorities

should be given on different levels within the organization. Normal pipeline runs of a security team should have higher priority than normal pipeline runs of regular DevOps teams. Configuring a priority policy would be a good solution. As already explained, prioritization constructs could be improved on all platforms, so no example is given here.

Test Shards

Some platforms—like CircleCI—have the option to "split" one task and divide the work. The execution of one task is instantiated several times, and work is distributed over multiple compute nodes. This is very efficient when performing tests. Assume that a regression test contains the execution of hundred individual tests. A normal task run executes these hundred tests sequentially. But the workload can also be spread over multiple instances of that task, such as in five instances of the same task, executing five times 20 tests in parallel, for example. Note that this puts a requirement on the test set. It must be possible to group tests and run them independently. This group of tests is called a *test shard*. The process to create the shards is called *test splitting*.

Creating test shards is possible in several ways. A simple algorithm just takes the hundred test cases and distributes them equally over five shards. The problem, however, is that you could end up with a shard containing only tests with a long test duration. A better approach is to divide the tests based on other characteristics. An optimized approach is to spread the test set over the five shards based on timing data. This is historic data based on previous test runs. After several runs, the ALM/integration platform has enough information to equally divide the tests efficiently over the task instances based on their duration.

Figure 5-4 contains three instances of the same test task. The total work of *Test_task_1.2.1* is spread over the three task instances, each executing 10 tests.

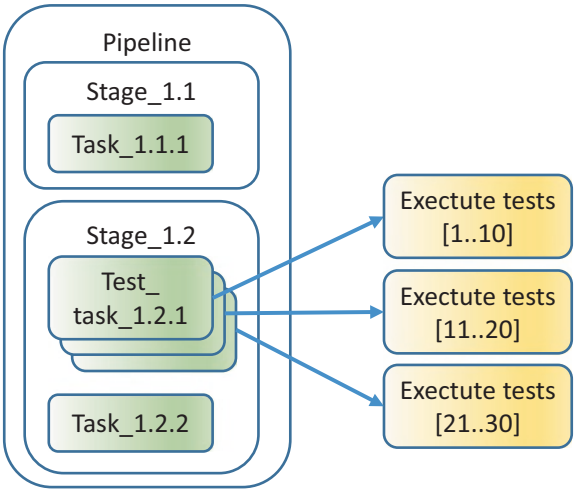


Figure 5-4. *Test splitting*

Templates and Libraries

It is possible to put all code in one big pipeline file and duplicate it in other pipelines if needed, but this does not improve readability or maintenance very much. By using isolated pipeline code, reuse is encouraged. Using templates or libraries is a way to move pipeline code to another file so it can be reused by other pipelines. *Template* in this context is a generic name. Some ALM/integration platforms offer the possibility to use some form of a template, but depending on the platform, the name and concept may be different. In Jenkins, for example, it is possible to use a shared library or use the load command to include a Groovy script in a pipeline.

In Azure DevOps, templates are used in the form of include or extend directives (see also Figure 5-5), which provides a lot of flexibility. Azure DevOps distinguishes two types of templates.

- *Extend templates*: The pipeline extends code defined in another file. This is called an *extend* template. An extend template works as a skeleton from which other

pipelines inherit its functionality. This allows the development of a generic pipeline structure, while details are implemented in each specific pipeline.

- *Include templates*: The main pipeline invokes templates to execute parts of the work. This is called an *include* template. An include pipeline includes pipeline code from another file in a certain section of the pipeline.

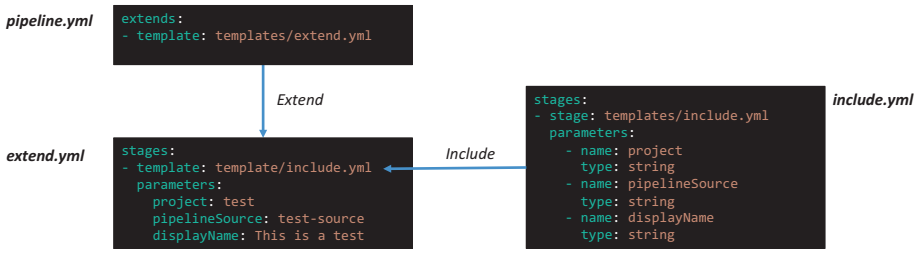


Figure 5-5. *Extend and include templates*

There are more options to add additional features to your pipeline. One example is adding pre- and post-jobs, using a feature called a *decorator* or *hook* (depending on the platform you use, of course). This makes it possible to add mandatory jobs from a compliance perspective. For example, a post job is added, which cleans up the workspace of an agent/node/runner and prevents files from remaining on the file system after the pipeline ended.

Gates and Approvals

A *gate* is an automated check to determine whether a pipeline is allowed to continue. This involves the validation of certain conditions, for example, the result of the *Analyze code* stage or the fact that a task is timed out. If the condition fails, the *gate* ends the pipeline execution.

An approval is a manual task that works similarly to a gate. The approval either allows the continuation of the pipeline or ends the pipeline execution and always involves a user who approves or declines. The *Perform dual control* stage is a typical example of an approval. Both gates and approvals are available on most platforms.

Workflow

Various platforms support pipeline declarations in which the functionality of the tasks and the workflow are intertwined. This makes it harder to distinguish functionality from workflow and makes it harder to understand the workflow of the pipeline. A good alternative is to separate the functionality from the workflow. Workflow becomes an isolated section of the pipeline declaration, which improves readability.

Listing 5-21 declares the workflow in a separate section of the pipeline. It does not include all the details of the jobs, but only their mutual relation and execution order. The unit_test and acceptance_test jobs are executed only after the build job has been finished. If both test jobs are completed, the deploy job kicks in.

Listing 5-21. CircleCI Workflow

```
workflows:
  version: 2
  build_test_deploy:
    jobs:
      - build
      - unit_test:
          requires:
            - build
      - acceptance_test:
          requires:
            - build
```

```
- deploy:
    requires:
      - unit_test
      - acceptance_test
```

Plugins and Marketplace Solutions

Plugins and marketplace solutions are a perfect way to add new features to ALM/integration platforms and pipelines. Plugins and marketplace solutions are available for various purposes, from a dashboard widget to a task that seamlessly integrates third-party tools with the ALM/integration platform. Depending on the platform, installing and using these plugins is straightforward, especially if they are self-contained. Some platforms, however, have the annoying habit that most of their plugins have dependencies and transitive dependencies with other plugins, often with specific versions. The plugins are not self-contained, which can cause dependency hell. But once you go through this struggle, plugins turn out to be powerful tools helping you to develop professional pipelines.

Repositories: Everything as Code

The life cycles of application code, infrastructure code (IaC), and pipeline code are often different. Does this mean that these types of code should be distributed over multiple repositories? As usual, it depends. Sometimes it is a matter of taste to distribute the different types of code over multiple repositories. Sometimes the CI/CD tooling forces the structuring of a project and its repositories, but a personal preference is to store application code, infrastructure code, test code, and pipeline code that belong to each other in one repository. Especially in the case of a microservice context, this makes sense. Everything that is part of a microservice is grouped because of the componentized character of a

microservice. And if more microservices are developed and more types of code, for example, security as code,[1] are added to the mix, repositories must be organized in such a way that everything is still easy to find and not scattered across various repositories.

In addition, if the different types of code are stored in the same repository, pipeline development becomes part of the teams' workflow and is more of a team effort.

Consider a situation where a DevOps team is responsible for the development of 15 very similar microservices. The team wants to make use of generic templates (or libraries), developed by another IT4IT team. What could a repository setup look like?

If all microservices are generic in nature, it makes sense to create one generic pipeline skeleton (base pipeline) that can be reused for all 15 microservices. Each microservice has its code repository containing the infrastructure code, the application code, and also the pipeline code. The pipeline of each microservice "inherits" from the base pipeline and adds specific features and variables. The pipeline also uses generic code, developed by another—IT4IT—team. If during development the DevOps team notices that some of their pipeline code has a generic character, they can decide to promote this code to a template library they manage themselves. A possible repository setup could look like Figure 5-6.

[1] *Open Policy Agent* gets more attention lately and fits nicely into the security-as-code domain (see [32]).

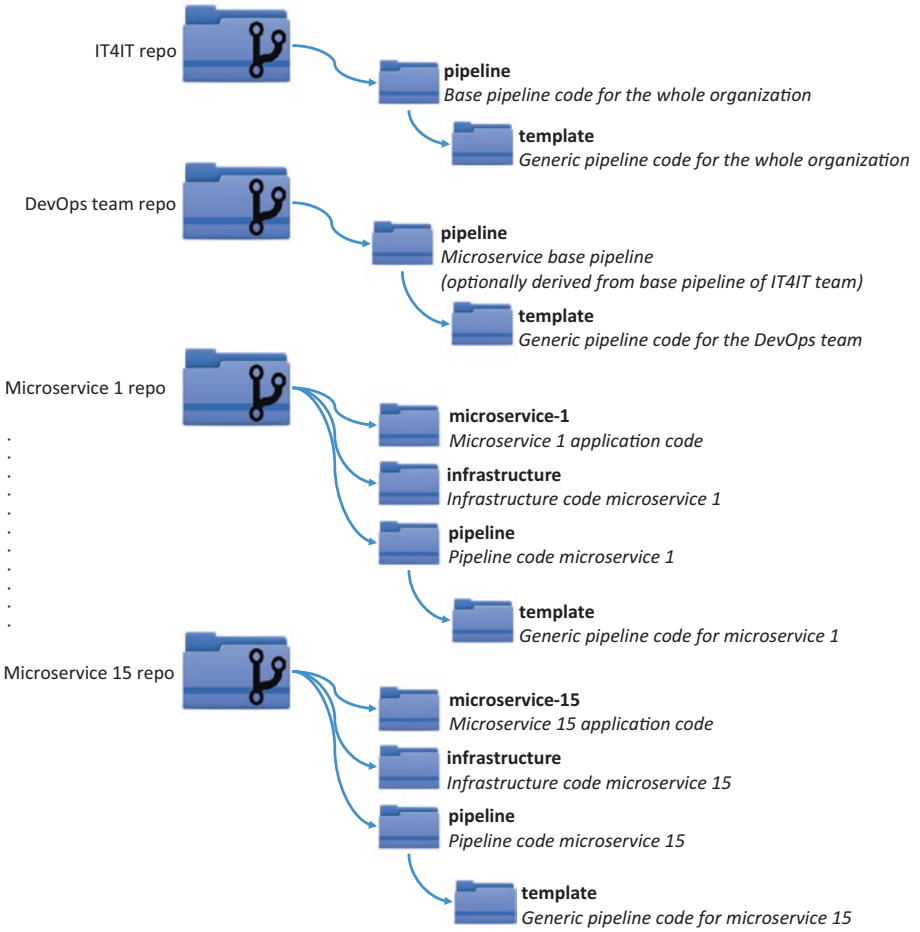


Figure 5-6. *Microservice repository setup*

Each microservice is contained in a repository, divided into an application code, infrastructure code, test code, and pipeline code section. The pipeline code section may contain multiple pipeline files. This section also contains a template directory with generic pipeline code, specific to this microservice. Generic pipeline code, developed by the DevOps team and used by all microservice pipelines, is moved to a separate repository.

Generic pipeline code, developed by an external—IT4IT—team, and used by all microservice pipelines, is also stored in a separate repository but managed by the IT4IT team.

Of course, you may decide to use a completely different repository setup, but the proposal in Figure 5-6 has been proven.

Note When adding pipeline code to the same repository as the application code resides, make sure that you have arranged a process to test the pipeline code properly. Untested or badly tested pipeline code results in a constant change of the code after it was merged into another branch. This also affects the application, because an incorrect working pipeline causes a stall in the software delivery process.

Third-Party Libraries and Containers

One of the security requirements listed in Chapter 3 mentions the fact that retrieving libraries and containers from the Internet must be done with great care. So, if a build task makes use of external libraries, make sure that retrieval of these libraries is secure.

Consider Figure 5-7. The pipeline retrieves data from various sources on the Internet. But what happens if one of these sources contains malicious code because the source is hosted by someone who does not have the best intentions? If access is permitted to use any source on the Internet, the pipeline may retrieve malicious code, include it in the artifact, and deploy it to production.

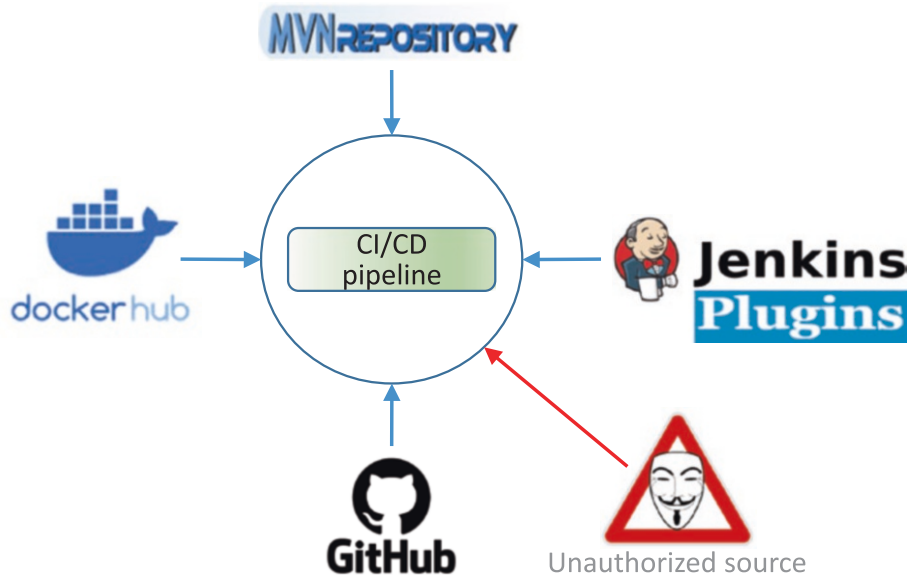


Figure 5-7. *Retrieve containers/libraries directly from external repositories*

At least one level of security should be considered. Assess which Internet sources should be authorized and only allow these sources to be accessed. Use a proxy containing a whitelist of the authorized sources. The pipelines are allowed to retrieve libraries and code only through the proxy. A procedure to add new assessed sources must be in place, of course; the setup must not become too rigid. Figure 5-8 shows a proxy layer denying access to the unauthorized source.

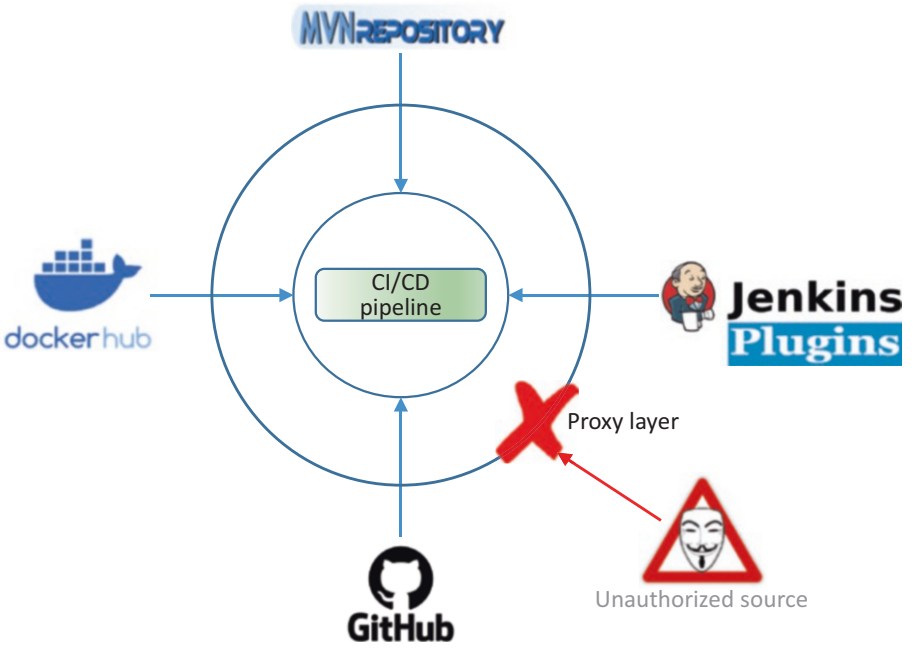


Figure 5-8. *Using a proxy to retrieve containers/libraries*

Another security layer can be added, using an internal repository, such as Artifactory or Nexus, or a repository positioned within the organization's data center or cloud account. The internal repository is refreshed with resources from the Internet, which are retrieved via the proxy. The advantage of using an internal repository is as follows:

- If external—Internet—locations are down, the pipeline still works because it only makes use of the internal repository.
- If resources on the external Internet locations are moved or not available anymore, the pipeline still works with the local copy.

- As part of the refresh action, in which the external resources are copied to the internal repository, a prescan can be performed on the resources, before they are internally exposed within the organization. Examples are as follows:
  - Malware and virus scan; resources copied from an authorized source may still contain malware, viruses, Bitcoin miners, etc. Scan them using a tool like Bitdefender.
  - Vulnerability scan; e.g., a base Docker image is scanned, and if it contains major vulnerabilities, it is put into quarantine.
  - Integrity scan; even if the Internet source is authorized, the copied resources may still be tampered with. The integrity scan makes sure that the downloaded resource is validated against a valid hash or digital signature.
  - Authorized IT products; check using Allow list and Deny list, for example, based on the Product Compliance List from [12].

A disadvantage of copying resources to an internal repository and prescanning them is that it is unknown up front which resources are used by a pipeline. Copying all resources and prescanning them takes a lot of storage and computing capacity. A practical solution is that pipelines use both an internal repository and a proxy for retrieving resources. Resources retrieved by the proxy must be scanned by the pipeline for malware, viruses, vulnerability, and integrity.

Other solutions make use of tooling with an extensive database that already performed prescans of a lot of packages and libraries. The tool prevents vulnerable packages and libraries from being downloaded in the

first place. A tool such as Mend Supply Chain Defender—formerly known as WhiteSource—can be used for this. Other alternatives such as Pyrsia (see [34]) make use of the power of blockchain to build trust for using open-source packages. Nexus Pro has the option to verify Pretty Good Privacy (PGP) signed artifacts.

Figure 5-9 shows the setup of an internal repository.

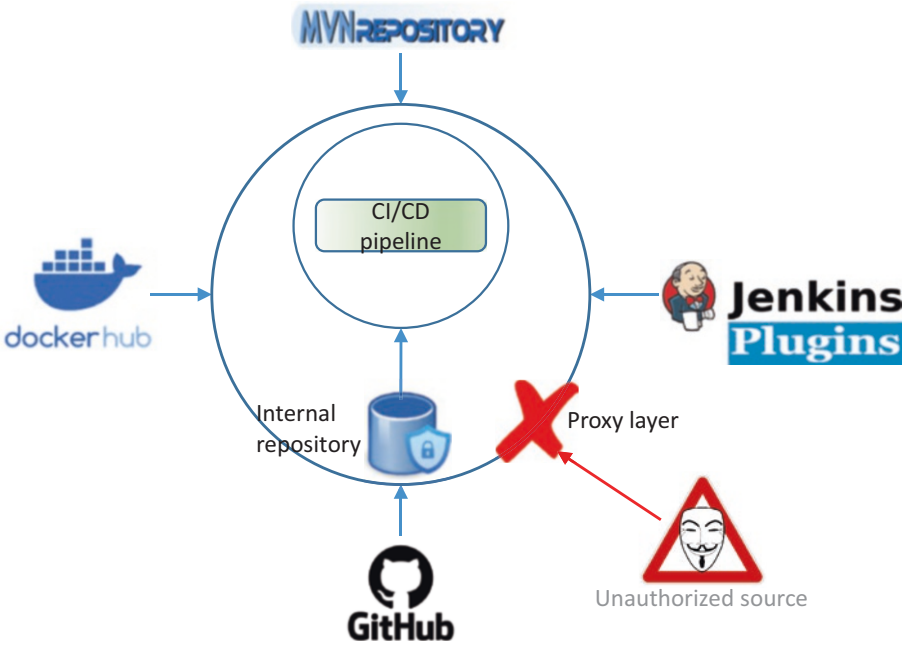


Figure 5-9. *Using an internal repository to retrieve containers/libraries*

Various ALM/integration platforms also support a pipeline cache or remote cache, which not only caches precompiled source files but also caches libraries and containers retrieved when building the artifact. If a remote cache is used, the setup looks like the one in Figure 5-10. The combination of the internal repository and the remote cache is complementary; the internal repository contains files used by the whole organization. The remote cache contains a subset of these files and is positioned very close to the pipelines.

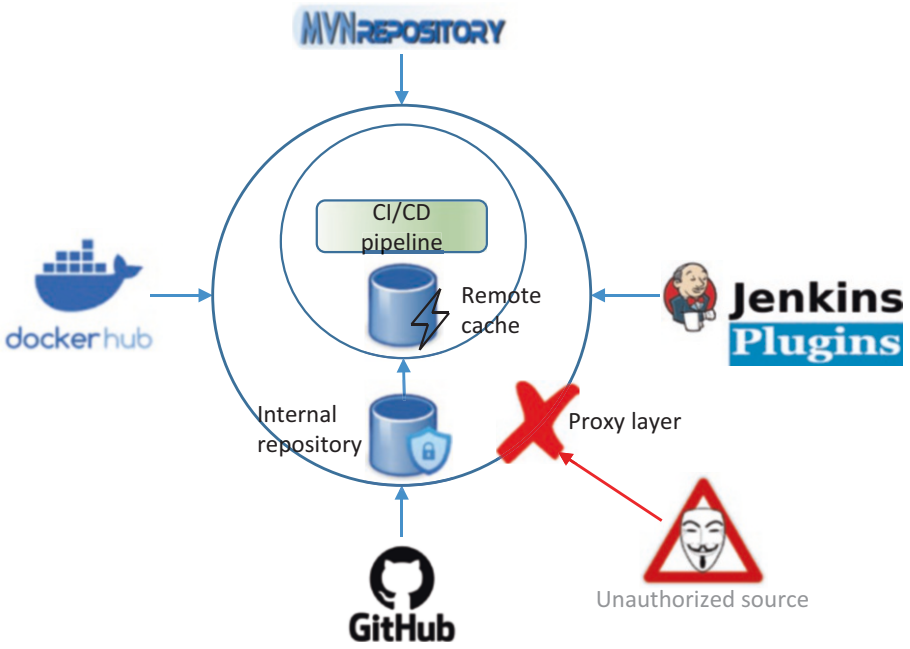


Figure 5-10. *Pipeline/remote cache and retrieval of containers/libraries*

Versioning and Tagging

A pipeline usually contains information that defines the state of a resource in the pipeline. *State* in this context means the status of the stages in the pipeline. A *resource* refers to any component used in the pipeline or the pipeline as a whole. Resources are work items in the issue tracker, application code, a build artifact, the pipeline running instance, etc. Together, these resources represent a certain state of a CI/CD process occurrence. A resource is represented using an identifier. For example, a release candidate of an application is represented by the application code, is identified using a commit hash, is associated with a work item ID, is built by the pipeline with a certain run ID, and delivers an artifact with a particular

version. Together these identifiers form a chain, which makes auditing the CI/CD process possible. The problem is often that these identifiers are different for each resource and tracing the chain of steps becomes difficult.

In this situation, tagging comes to the rescue. Tagging is adding a piece of information to a resource. Tagging can be seen as adding metadata to describe the state of a resource, and it helps in implementing the requirement "All changes are traceable."

In the ideal world, it would be perfect to tag every resource that contributed to the creation, deployment, and test of a release artifact. In practice, however, tagging is often restricted to only a subset of these resources.

Versioning makes it possible to identify the different states, and you can use tagging as a way to version a resource. This means a tag can be in the form of a version, but it doesn't have to be one. A popular versioning format is the "Semantic Versioning" scheme, which defines the major, minor, and patch versions. The format is MAJOR.MINOR.PATCH, for example, version 2.3.1 (see also [19]).

Assuming a team wants to apply tags in the form of a semantic versioning scheme, they first need to determine which resources support tagging. If a resource does not support tagging, there may be other ways to identify the state of a resource, for example adding the version to the name of the resource. Using a version in the filename of a build artifact is such an example. What does this mean for the pipeline design? Consider the following case:

A team uses Jira as their issue tracker system and Git as a source control management system. A Git commit represents one Jira ticket, which has to be provided in the commit message. The team uses the Feature branch workflow. Jenkins is used to build and deploy the artifact—an AWS Lambda app—to an AWS account. Artifacts are stored in Sonatype Nexus.

Semantic versioning is used and tagging is applied only in case an artifact is built from the main branch. The team wants to tag as many resources as possible. The tag must contain the release version.

Given this setup, the following are possible actions to be taken in the pipeline:

- Tag the Git commit with the release version (for example, `git tag -a v2.3.1 9fceb02`).
- The Git commit message contains a reference to the Jira ticket if the commit is pushed to the repository.
- Add a label to the Jira ticket with the release version. A Jira REST API is used to create this label.
- Add the release version to a Jenkins build by setting the release version in the job display name.
- Add the release version to the artifact filename in Nexus. Tagging is not needed if the artifact name already contains the version, but it is possible to add a tag with the Nexus Pro version.
- Tag the AWS Lambda or the AWS Stack with the release version.

The design of the pipeline of the main branch of a feature branch workflow is extended with tagging tasks, resulting in the BPMN model shown in Figure 5-11.

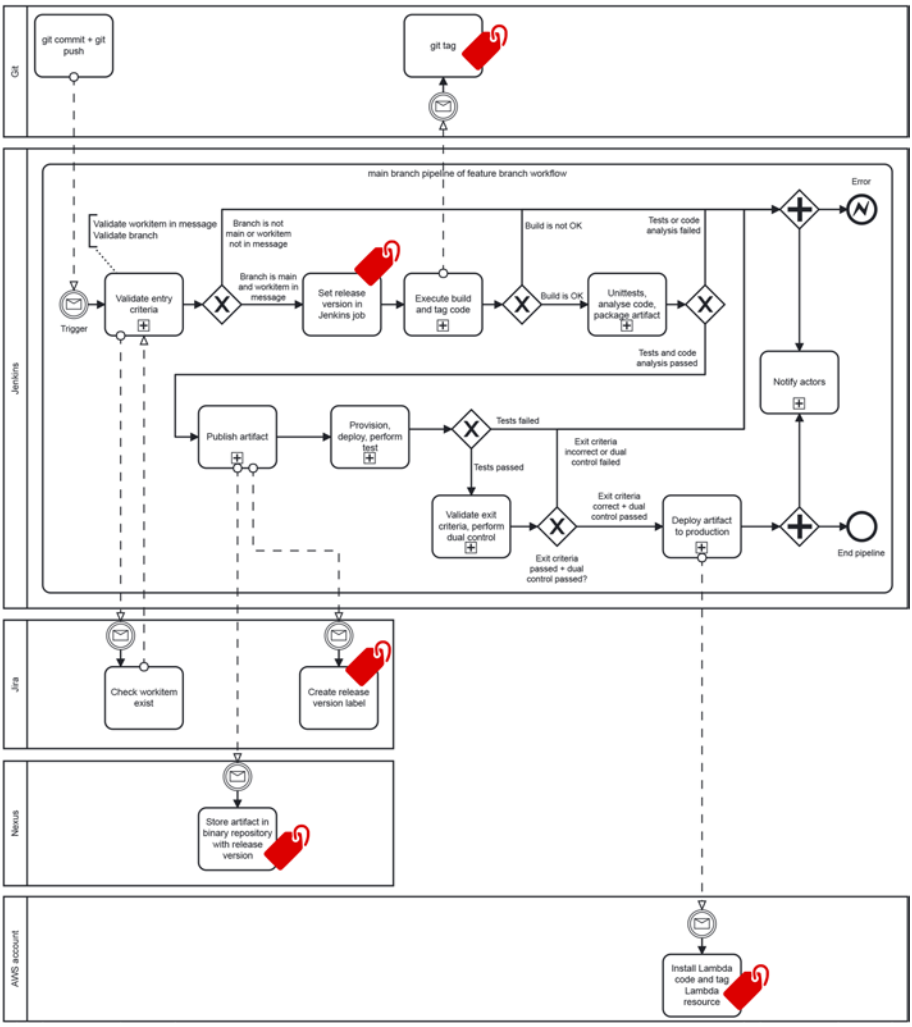

Git
git commit + git push
git tag
Jenkins
main branch pipeline of feature branch workflow
Error
Validate workitem in message
Validate branch
Branch is not main or workitem not in message
Build is not OK
Tests or code analysis failed
Trigger
Validate entry criteria
Branch is main and workitem in message
Set release version in Jenkins job
Execute build and tag code
Build is OK
Unittests, analyse code, package artifact
Tests and code analysis passed
Notify actors
Publish artifact
Provision, deploy, perform test
Tests failed
Tests passed
Exit criteria incorrect or dual control failed
Validate exit criteria, perform dual control
Exit criteria correct + dual control passed
Deploy artifact to production
End pipeline
Exit criteria passed + dual control passed?
Jira
Check workitem exist
Create release version label
Nexus
Store artifact in binary repository with release version
AWS account
Install Lambda code and tag Lambda resource

Figure 5-11. *BPMN, versioning and tagging*

Note Tagging is done in several stages in the pipeline. Setting the release version in the Jenkins job is one of the first things done in the pipeline, but the *Execute build* may still fail. But this is not a problem.

The pipeline intends to create a release candidate with a certain version that can be deployed to production, but the pipeline job may still fail along the way. Maybe some resources are already tagged, and others aren't. Tracing the tags in the chain reveals that the pipeline stopped at a certain stage and the artifact was not deployed to production.

Environment Repository

A well-developed application does not contain any environmental properties. The artifact must be built once but must be able to run anywhere. Environmental properties must be added during deployment, for example, by enriching placeholders in a property file with the correct data during deployment. Data such as database credentials or HTTP endpoints are stored in an environment repository, and as soon as a deployment starts, the placeholders in the property file are replaced with the database credentials and HTTP endpoints associated with the target environment to which the application is deployed.

There are different types of environment repositories. The type of environment repository to use also depends on the security classification of a certain property. Database credentials have a higher risk rating than an HTTP endpoint, so database credentials should be stored in a more secure environment repository. Here are some examples:

- *Variable in the pipeline*: The simplest solution is to just define properties as (conditional) variables in the pipeline code itself. During deployment, the target environment is determined, and a specific set of variables is used. This solution is easy to implement.

A disadvantage is that the pipeline code cannot contain sensitive information, and updating a variable means that the pipeline code must be updated.

- *Storage on a file system or SCM repository*: Properties are stored on the file system or in a repository—such as Git—and the files are arranged per target environment; `dev.test.properties`, `system.test.properties`, `acceptance.test.properties`, and `production.properties` are a few examples. During deployment, the target environment is determined, and the property file associated with this environment is determined and included in the deployment. A disadvantage is that a property file stored on a file system or a code repository cannot contain sensitive information. The first layer of security can be established in such a way that access to the files is only allowed by the pipeline and by engineers of the DevOps team. Other people are not allowed to access the filesystem or repository.
- *Secret management tools*: There are several (open source) secret management tools that help with content encryption of files in an SCM (Git). Examples are SOPS and Blackbox. See [37].
- *Integrated environment repository*: Some ALM/integration platforms already have an integrated solution for storing environment properties, with names such as *Library, Config Store Service*, or *Credentials store*. It is a repository in which properties—confidential or not—can be stored. Some of these platforms also offer the possibility to store complete files. The properties and files are encrypted.

The encryption and decryption keys are managed by the platform. This only leaves the question of whether the storage of the keys is secure enough. In the case of Jenkins, keys are stored on the file system on which Jenkins is installed and can be accessed only by the Jenkins user (in the case of Linux). For SaaS solutions, the provider of the solution manages the encryption keys.[2] This solution works fine for medium and low-security classified properties.

- *Vault*: For really high-security classified information such as database credentials, it is best to use a vault. On some ALM/integration platforms, the integrated environment repository is backed by a vault. The next section elaborates a bit more on vaults and secrets management in general.

Secrets Management

As mentioned in Chapter 3, secrets—passwords, tokens, keys, credentials—used by an application preferably must be stored in a vault. This can be Azure Key Vault, AWS Key Management Services, AWS Secrets Manager, HashiCorp Vault, or a Hardware Security Module (HSM). Important to consider is where the secret is created and how it can be used by the application. Is the source location of the secret the same as the target location? Or in other words, is the secret created in the location where it is also used by the application, or is it created somewhere else and must it be transferred to another destination so the application can use it? This also raises the question of whether the source and target

[2] Unclear, however, is whether these encryption/decryption keys are specific to one tenant or whether they are used across tenants.

locations both meet the secret's security classification and whether the transport from the source to the target location is secure enough. Cases exist in which vaults are not used for whatever reason or the secret cannot be created in the vault itself and it has to be manually transferred from the source location. Different situations are possible. Let's go through some options, in order of most secure to less secure:

1. The safest solution is that the target platform in which the application runs also manages the secret. The target platform creates the secret in a vault, and the vault maintains its life cycle (see Figure 5-12). No pipeline is involved. This is a safe way to deal with secrets because the secret is not exposed and may even never leave the vault. Key rotation is managed by the vault by which the key is automatically renewed.

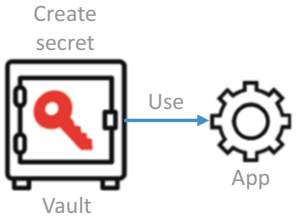


Figure 5-12. *The target platform creates a secret in the vault*

2. Often, the vault does not "know" it needs to create and manage a secret. A pipeline is required to trigger the creation of the secret in the target vault. This means that the vault already has functions to create the secret and the pipeline only executes these functions. The application can use the secret directly from the vault or uses the vault's built-in

functions to perform an action—e.g., signing data—that makes use of the secret in the vault. In addition, the pipeline triggers key rotation, which is managed either by the pipeline or by the vault. Figure 5-13 shows this setup.

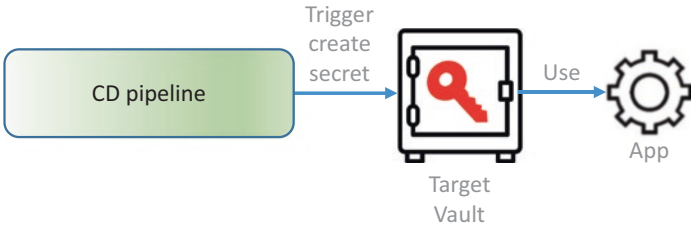


Figure 5-13. *The pipeline triggers the creation of a secret in the vault*

3. The secret is already precreated by another (source) system and has to be transferred by the pipeline to the target vault. The source location can be a vault again or another system that manages the secret. It can also be a system that uses a vault as its secret provider. The transfer of the secret—using a pipeline—from the source to the target vault is fully automated and secured. Secure transfer measures may include mTLS and/or even digitally signing the secret. The secret is not stored in the integration platform. Members of the DevOps team are not able to view the secret's value. The pipeline should never expose the secret in logs or any other way. This process is depicted in Figure 5-14.

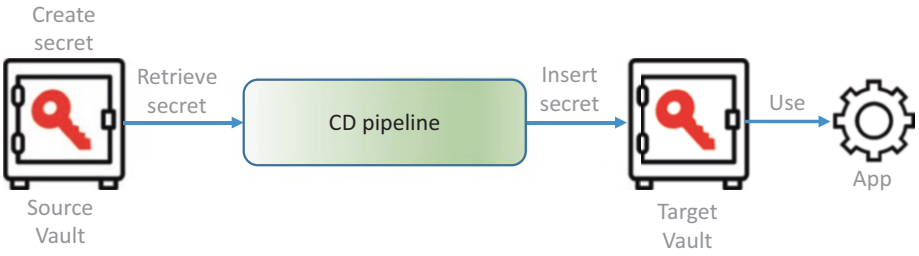


Figure 5-14. *Transferring a secret from the source to a target vault*

4. The retrieval of the secret from the source cannot be automated. Maybe the source location is not even a vault. This means that a DevOps engineer has to log into the source system, extract the secret, and store it in the ALM platform. Some of the ALM platforms support the option of storing secrets, as variables or in a secret file. The pipeline retrieves the secret from the ALM platform and inserts it into the target vault. Figure 5-15 shows this process.

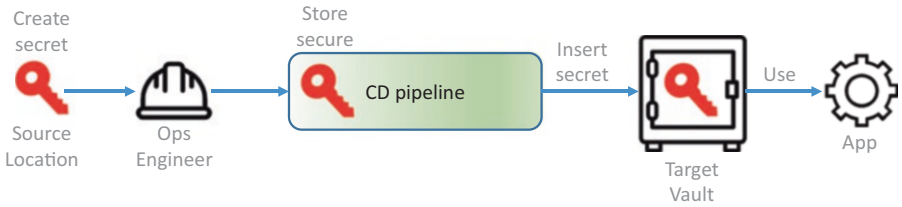


Figure 5-15. *Manual transfer from source to target vault*

5. The destination is not a vault. The secret must be "injected" directly into the application or deployed as a file accompanied by the application. This is depicted in Figure 5-16.

Figure 5-16. *Manual transfer from source and "injecting" the secret in the artifact*

Although this setup is not preferred from a security point of view, it is used a lot. One of the next paragraphs explains which security issues are involved with this solution.

Database Credentials

The secrets management cases in the previous paragraph are a bit abstract, and a little more clarification seems in order. Consider the credentials of a database. In Figure 5-17, the database is situated in a highly managed infrastructure, such as a cloud environment. The pipeline calls an API of the vault, which acts as an identity provider of the database and creates the database secret (credentials). Because the app has a trusted relationship with the vault, it is allowed to use the database secret to access the database. The vault is responsible for the rotation of the database secret.

The responsibility of the pipeline is limited. After the initial trigger to create the database secret, the system—consisting of a vault, an app, and a database—manages and uses the database secret. This is a secure solution because the secret in the vault is accessible only by a trusted party: the app. This trust is based on security policies and other infrastructure measures.

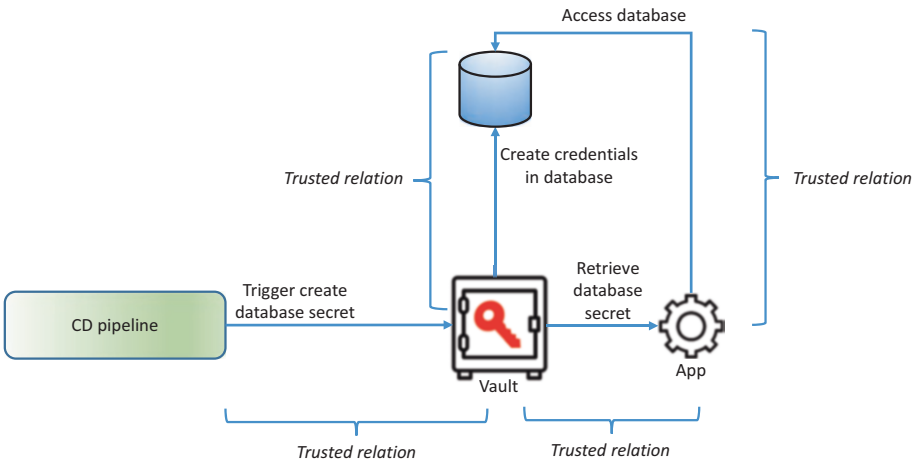


Figure 5-17. *Database secrets in a highly managed environment*

The second example, depicted in Figure 5-18, is a setup in which the database secrets (credentials) are generated in the database by an Ops engineer or database administrator (DBA), who transfers the secret to secure storage on the ALM/integration platform. As part of the app deployment, the pipeline contains a task that "injects" the secret into the app, after which it is deployed to the target environment. The "injected" secret is used by the app to access the database. Secret rotation is triggered by the Ops engineer who starts the whole process again.

The pipeline has some more responsibilities compared to the previous example. The secret is stored in secure storage and must be retrieved by the pipeline. The pipeline injects the secret into the app, after which the app is deployed to the target environment. This example, however, suffers from various attack surfaces.

- There are trusted relationships between the ops engineer and the database, and the ops engineer and the secure storage. From a security perspective, this is a very weak point in the chain. Humans cannot be trusted completely.

- Storing the secret in the ALM/integration platform is less secure unless the secret storage is a vault. For a lot of platforms, this is not the case.
- Injecting a secret in the app is an example of bad engineering, but sometimes these things occur. One of the security issues is, that from that moment, the secret is stored in a less secure place, namely, the app.

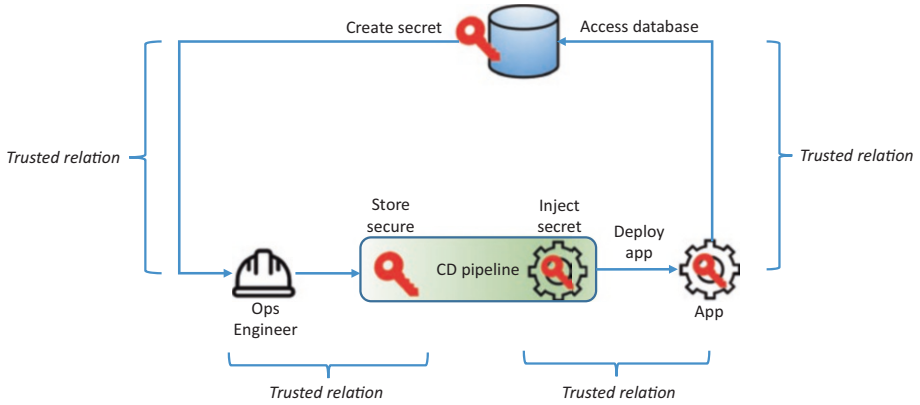


Figure 5-18. *Manual transfer of database secrets*

Feature Management

Most developers know what a feature flag is. A basic feature flag is an `if` statement that determines whether a function in the code is executed or not. More complex feature flags make it possible to disclose a certain function only for a specific user group and/or target environment. This is also the power of using feature flags; functions that were previously hidden because they were in an experimental state, for example, can be enabled with the click of a mouse. This makes feature management

a good alternative for an A/B testing strategy using a canary or blue/green deployment. In addition, feature flags can also be used to keep the mainline of the code in a stable state. Unfinished features in the mainline are hidden in a production environment.

Java developers may be experienced in implementing feature management with the use of Spring Cloud Config, but it is interesting to see that several ALM/integration platforms also begin to offer feature management.

Feature management allows more control over feature flags in pipelines and beyond. Toggling features on or off can be done at different stages in the software supply chain. Figure 5-19 visualizes the possibilities.

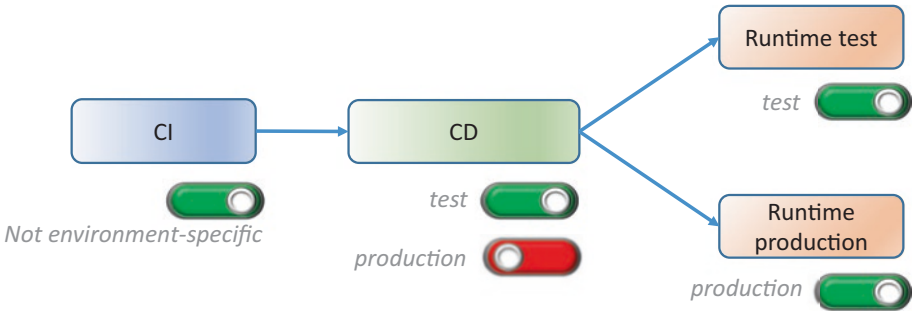


Figure 5-19. *Toggle feature in CI, CD, and runtime stages*

- *Build (CI) stage*: If a feature flag is toggled in a build stage, the flag becomes part of the artifact, and irrespective of where the artifact is deployed, the value of the flag (true or false) determines whether a certain function is enabled. Setting feature flags in the build (CI) stage is static and a bit rigid. If a disabled function must be enabled, a new artifact version must be built.
- *Deploy (CD) stage*: An alternative is to use feature flags in the deploy (CD) stage. The value of the feature flag is "injected" in the artifact at deployment time, which

makes it possible to enable a function for one target environment and disable it for another environment. In Figure 5-19, a feature flag is enabled on the CD level for the test environment and disabled for the production environment.

- *Runtime*: Although using feature flags in the Deploy (CD) stage provides a bit more flexibility compared to the Build (CI) stage, it can be improved even more. Modern feature management makes it possible to use feature flags in a runtime environment. Functions that were disabled in the runtime environment can be enabled dynamically, for all users or a selected group of users, without the need to rebuild or redeploy a new instance of the application.

Implementing feature management in this way does pose some constraints to the way the code is developed. The user interface of the feature management system makes it possible to toggle a feature with the click of a mouse. The application code, however, must be able to interact with the feature management system to make that happen. This is also one of the drawbacks. The application code needs a third-party library and includes additional statements from that library. This results in some intrusive code in the application. Fortunately, this code can be removed again if the function becomes available to all users. In addition, using feature flags in a runtime environment also requires a connection between the feature management system and the application. The application is packed with an SDK that polls APIs of the feature management system. The APIs are used to synchronize between the SDK and the feature management system. The SDK can detect the state of the feature flags in the feature management system and use them at runtime. You need to make sure that this connection is secure.

Figure 5-20 shows how feature flags can be enabled and disabled—also for a particular user group and/or environment—in GitLab.

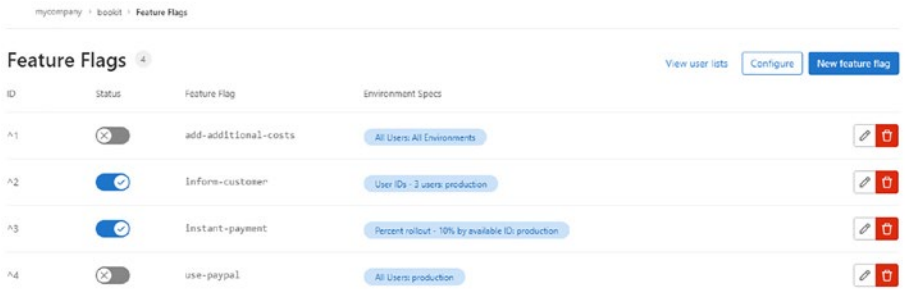


Figure 5-20. *Gitlab, feature flags*

Listing 5-20 contains the `if` statement with a feature flag called `add-additional-costs`. It makes use of the feature management system Unleash (see [24] for more information).

Listing 5-20. Unleash/Java Example of a Feature Flag

```
if (unleash.isEnabled("add-additional-costs")) {
    // Additional costs are calculated and added to the booking
} else {
    // The booking is processed without additional costs
}
```

Development in the Value Streams

CI/CD development is a container concept that includes aspects that deal with the automation of the software supply chain. This varies from setting up the ALM/integration platform to the actual realization of a specific pipeline. As explained earlier, activities related to CI/CD are present in different *value streams* and divided over numerous teams and organizational units. Development takes place on several levels within

the organization. The following development topics provide some insight into what kind of development and at what organizational level the responsibilities lie.

- *CI/CD SaaS solution*: A CI/CD SaaS solution is an ALM/integration platform developed by an external company. It can be configured using add-ons or plugins.

Responsibility for the Central Organizational Unit	Responsibility for the DevOps Team
A specific organizational unit, like an IT4IT team, manages the use of the SaaS solution for the whole company and is also involved in the integration and additional development of add-ons/plugins. However, the management and development of the SaaS platform itself is the sole responsibility of the provider of the platform.	Individual DevOps teams are usually not involved with the management of SaaS solutions or the development of specific add-ons or plugins.

- *Platform infrastructure development*: Instead of using a SaaS solution, developing your own (reusable) integration platform using IaC is another option. The result is code, which is developed once and used to roll out the complete integration infrastructure. One example is, for instance, a Docker container containing a completely integrated setup with Jenkins, InfluxDB, Grafana, etc.

Responsibility for the Central Organizational Unit	Responsibility for the DevOps Team
This activity is typically done on a higher organizational level (by an IT4IT team) because it is costly and requires specific knowledge.	Although DevOps teams sometimes develop their own integration infrastructure code, this is not recommended. It also depends on the type of organization.

- *Platform infrastructure hosting*: This involves the actual provisioning of the integration infrastructure and the hosting. It does not involve much development, but it does include the configuration of the hosted infrastructure.

Responsibility for the Central Organizational Unit	Responsibility for the DevOps Team
A valid use case is a centrally hosted integration platform, managed by a specific organizational unit. The platform is shared with multiple DevOps teams.	The integration platform code can also be developed (once) by a specific team, while each DevOps team makes use of it and manages the hosting.

- *Development of a base pipeline*: Development of a base pipeline means that the pipeline code itself is developed once and can be reused by different DevOps teams. These pipelines are configured as desired.

Responsibility for the Central Organizational Unit	Responsibility for the DevOps Team
It makes sense that a specific IT4IT team develops such a base pipeline.	DevOps teams make use of the base pipeline and configure it according to their needs.

- *Development of generic templates libraries*: If certain pipeline features are used often by multiple DevOps teams, it makes sense to develop them as a template or a library that can be (re)used by multiple DevOps teams.

Responsibility for the Central Organizational Unit	Responsibility for the DevOps Team
A specific IT4IT team develops these templates/libraries.	DevOps teams make use of the generic template/library in their pipelines.

- *Pipeline code analysis and compliance scanning*: Because pipelines are just code, they can be scanned on code quality and validated whether the pipeline is constructed according to organizational policies.

Responsibility for the Central Organizational Unit	Responsibility for the DevOps Team
There are plenty of code analysis tools to integrate into a pipeline and analyze the applications' code, but the tools that analyze the pipeline code itself are rather scarce. A specific IT4IT team is required to develop this kind of tooling.	This is usually not something a DevOps team itself does because that would be a bit like a fox guarding the henhouse.

- *Development of specific templates/libraries*: If certain pipeline features are used in multiple pipelines within one DevOps team, it makes sense to create a template/library from it to prevent redundancy of code. The templates/libraries are usually not shared with other teams.

Responsibility for the Central Organizational Unit	Responsibility for the DevOps Team
This is specific for DevOps teams themselves, so no central team is involved.	The responsibility lies within the DevOps team.

- *Development of pipelines*: This concerns the development of pipelines used by DevOps teams.

Responsibility for the Central Organizational Unit	Responsibility for the DevOps Team
This is specific for DevOps teams themselves, so no central team is involved.	The responsibility lies within the DevOps team.

Application development is done by an engineer developing code on their local machine, performing unit tests, and, when finished, pushing code to a source code management system. This triggers a pipeline on an ALM platform or integration server, which builds, deploys, and tests the application.

Throughout this book, the parallel is drawn between pipeline development and application development, so applying the same principles to pipeline development means that a developer develops the pipeline code, performs the unit tests on the pipeline code, and, after completion, pushes the pipeline code to the source code management system, which triggers…a pipeline. This introduces the concept of a pipeline of pipelines: a DevOps assembly line in which pipelines are built, deployed, and tested using another pipeline.

Let's elaborate a bit more on this and see where this leads.

Simplified Pipeline Development

Application development has been done the same way for a long time. The code is created, probably using plugins installed in the integrated development environment (IDE), to analyze the code for vulnerabilities, code quality, performance issues, etc. In addition, unit tests are created and executed within the IDE. If everything is fine, the application code is committed and pushed to the remote server.

Pipeline development at its simplest is when a developer creates the pipeline code in their favorite IDE. Pipeline code involves one or more files. Local testing is hard. The developer does not have a local ALM/integration platform installed or can make use of a test platform, so the pipeline code is developed, barely tested, and pushed to the repository; after that, the developer hopes for the best. This is not a very optimal way of working, but this does happen a lot.

Figure 5-21 schematically shows how this process works. The developer creates or updates the pipeline code—for example, in a feature branch—and pushes it to the remote repository when finished. This repository also contains the application code. However, the pipeline code is not unit tested at all.[3] As soon as the pipeline code is pushed, it starts executing, but it was never tested properly, so errors and bugs are to be expected. That is not a desired workflow, is it? Are there ways we can do this a bit better? Well, actually we can.

[3] Sometimes some form of testing may be possible using a dry-run flag (like `mvn release:prepare -DdryRun=true`), but it is still a hacky way of testing the pipeline code.

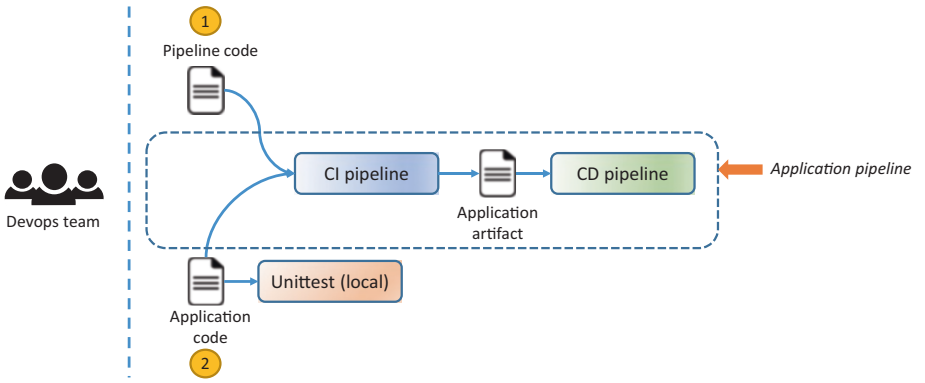


Figure 5-21. *Simplified pipeline development*

Extended Pipeline Development

A bit more sophisticated way is to perform unit tests on the pipeline. How this can be done is explained in the next chapter that deals with testing pipelines, but in essence, the developer has a pipeline test environment used for development, in which the unit tests of the pipeline are executed (see Figure 5-22). This approach gives more confidence that the pipeline code is of decent quality. Preferably this test environment is a local environment, for example, a Jenkins instance installed on the developers' local workstation.

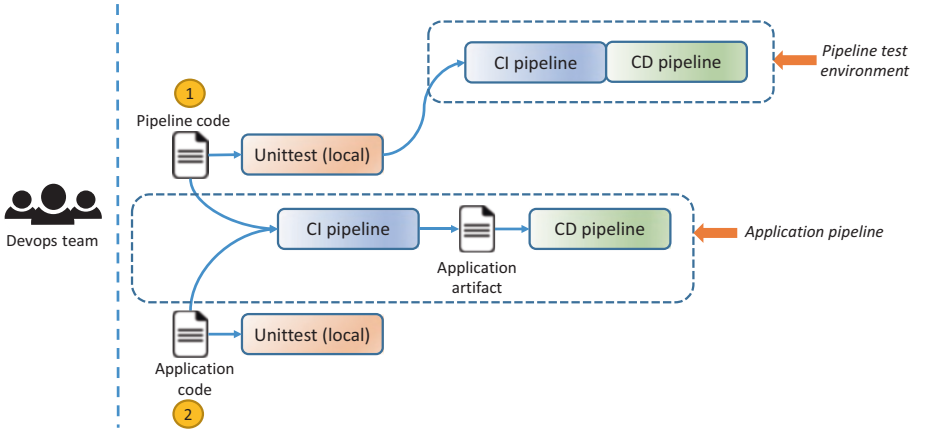


Figure 5-22. *Extended pipeline development*

Advanced Pipeline Development

The *extended pipeline development* method can be raised to the next level in which the pipeline is not only unit tested but also undergoes all—or at least some—stages of the Generic CI/CD Pipeline itself. In this development method, the pipeline of pipelines concept is applied to the full extent (see Figure 5-23). Later in this chapter, the stages of the pipeline of pipelines are explained in more detail.

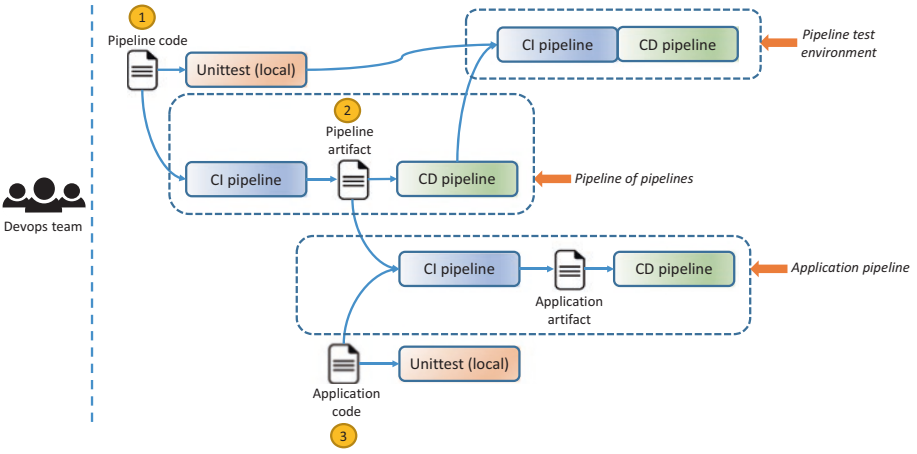


Figure 5-23. *Advanced pipeline development*

This development method distinguishes three important phases in the development process.

- Pipeline code is developed on a local machine after which unit tests are performed. Preferably, unit tests are performed on a local instance of the ALM/integration platform.
- After pushing the pipeline code to the repository, it is processed using an assembly line for pipelines. This assembly line performs similar stages as in the

case of application development but is now focused on pipeline code. This assembly line of pipelines is referred to as the *pipeline of pipelines*.

- The output of this pipeline—the a*pplication pipeline*—is a thoroughly tested pipeline artifact; this pipeline is used to build, test, and deploy the application.

Note All three phases involve an integration infrastructure. This can be the same ALM/integration platform used for running the application pipeline—on which storage and processing of each phase are separated—but it is also possible to use three different physical infrastructures.

The advanced pipeline development approach has some drawbacks. The team has to set up everything itself. Imagine an organization having 500 teams; this would not make any sense. It is too costly, it takes too much time, and in addition, not all teams have the expertise to develop something like this.

In essence, the approach is good, but some of the work needs to be centralized and moved to a dedicated IT4IT team. The IT4IT team develops the tools and infrastructure of the pipeline of pipelines. DevOps teams make use of it.

Develop a Base Pipeline

In the previous examples, the DevOps team developed the application pipeline. An alternative is to use a base pipeline that has been developed by a central IT4IT team. The base pipeline contains default stages and tasks and is used for a specific context, for example, a Java/Maven/Linux context or a Python/Windows context. The base pipeline contains some

mandatory tasks that should not be overwritten. The DevOps team extends its pipeline from the base pipeline and configures it to its needs, so it can be used to build, deploy, and test the application.

Creating the base pipeline requires specific knowledge, but centralizing the development can save a lot of time and money in the end. Creating a base pipeline also allows enforcement of certain policies or security restrictions, which become automatically part of the extended base pipeline.

In Figure 5-24, the base pipeline is tested by the IT4IT team—also making use of a pipeline of pipelines—and the resulting base pipeline artifact is centrally stored and can be used by the DevOps teams. Of course, after extending and reconfiguring the base pipeline, the DevOps team can also perform (unit) tests of their pipeline to make sure that it works as expected.

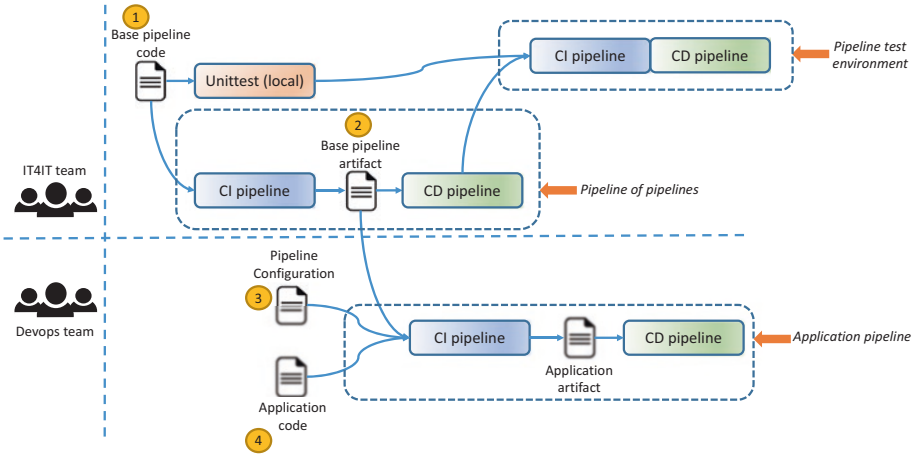


Figure 5-24. *Base pipeline*

Pipeline Generation

Extending a base pipeline has its limitations. What if the team needs a completely different pipeline or deviates from the base pipeline so much that using it is not justified? Instead of creating a base pipeline, the pipeline used by the DevOps team can also be generated using a pipeline generator. The feature richness of such a pipeline generator varies from creating code snippets, which need to be assembled by the DevOps team, to the generation of a complete customized pipeline that undergoes the stages also used in regular application-oriented pipelines. The input of a pipeline generator is a repository managed by a DevOps team. The pipeline generator scans this repository, detects the configuration, and starts the creation of artifacts (pipeline code and testware). Figure 5-25 depicts a setup with a pipeline generator.

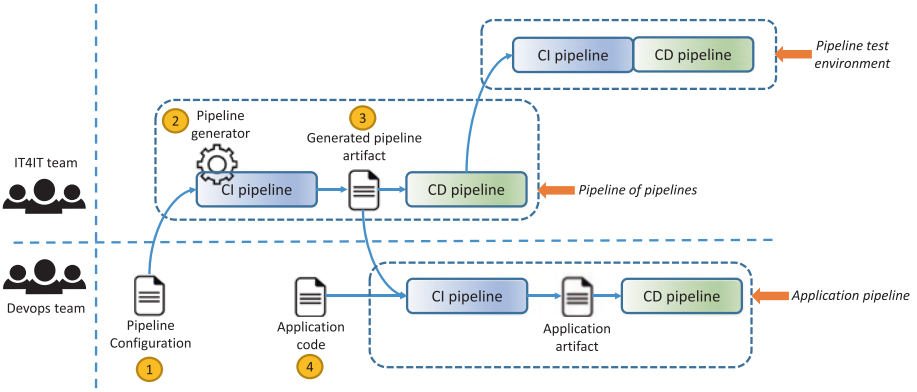


Figure 5-25. *Pipeline generator*

These kinds of tools, however, are scarcely available, and if there are any commercial tools out there, they are not well-known. Until then, it looks like organizations have to develop these tools themselves. Usually, this is a task of an IT4IT team dedicated to this job, but to prevent the "not invented here" syndrome, a cooperation model with DevOps teams is

needed. In this case, an innersource model seems a good fit. Innersourcing allows DevOps teams to help with the creation of the tooling. If a DevOps team has a good idea, it can start working on that idea and, when finished, create a pull request. The IT4IT team approves the pull request (or not), and the code is merged into the codebase of the pipeline generator.

The pipeline generator itself is embedded in a pipeline of pipelines, in the *Execute build* stage. The produced artifacts consist of pipeline code and testware. These artifacts are deployed to a "test" integration server/ALM platform, where they are tested.

The pipeline generator itself is also an application. The development steps of this tool are not included in the figure.

Of course, this all sounds nice and sophisticated, but essentially a pipeline generator is a complex piece of software that takes a long time to develop. And what does such a tool look like? What are the requirements? Let's make a small attempt to make it a bit less abstract.

- The stages of the Generic CI/CD Pipeline are used as a pipeline blueprint. Based on the given DevOps teams' branching strategy (trunk, feature branch, Gitflow, …), the flow of the pipeline is constructed.
- Configuration of the pipeline generator must be as simple as possible. The tool retrieves most of the information by scanning the code repositories of the DevOps team.
- Stages, tasks, but also testware to test the application are automatically constructed based on scanning the repository. Here are some examples:

    - Based on a given repository, the repository is scanned for application code, and based on that, a pipeline with an *Execute build* stage is constructed. For example, if the tool finds a `pom.xml` file in the repository, it likely is a Maven project and constructs a Maven build task in the pipeline.
    - If the pipeline generator finds Postman collections, a *Perform test* stage is created, and test tasks are added to this stage; e.g., the task assumes that a tool such as Newman is used to execute the Postman collections in the pipeline.
    - Similarly, if the tool finds Cucumber tests (e.g., based on `*.feature` files it finds in the repository), it constructs a test task and adds it to the *Perform test* stage.
- The tool must contain a library of prefab stages and tasks, which are configurable by the DevOps team. These stages and tasks also include unit tests.
- Based on corporate policy, mandatory stages and tasks are added to the generated pipeline. This also means that a DevOps team is not allowed to delete them when using the generated pipeline.
- The tools contain additional features, which are added to the generated pipeline, for example, tagging, generation of a release note, and notifications to specific communication channels.
- DevOps teams must be able to add specific tasks for their case.

- DevOps teams must be able to add new, reusable generic tasks to the prefab library. These tasks can be used again by other DevOps teams.
- The pipeline of pipelines is maintained by an IT4IT team.

This list of requirements is nonexhaustive, and organizations can make it as complex, extensive, and feature-rich as they want.

Note Unfortunately, I never had the chance to develop something like this, but it is not a completely crazy idea. Some organizations did develop a pipeline generator, and I've seen examples of it, created by colleagues.

Pipeline of Pipelines (DevOps Assembly Line)

A few of the pipeline development methods described in the previous paragraphs have one thing in common. The created pipeline code undergoes some processing stages similar to application development. Pipelines are built, deployed, and tested using a pipeline assembly line, the *pipeline of pipelines*. Depending on the pipeline development method and the platform used, the implementation of the pipeline of pipelines may differ, but it has similarities with the Generic CI/CD pipeline, as shown in Figure 5-26.

- *Trigger*: After a developer has developed the pipeline code, changes are committed locally, and preferably pipeline unit tests are performed. If the developer is confident about the pipeline code, they push the code to the remote server. This triggers the pipeline of pipelines.

- *Validate entry criteria*: The input of a pipeline of pipelines is either pipeline code or a pipeline declaration file. One of the entry criteria is to determine whether the file or set of files meets certain requirements. For example, if the input is a YAML file that defines the pipeline, the entry criterion is that it must be a valid YAML file, to be checked using a tool such as yamllint.
- *Execute build*: Pipelines are usually files, containing scripts or containing pipeline declarations. These types of files are interpreted and used as is, so a compilation of a pipeline artifact is not needed. The *Execute build* stage can be omitted in these cases. In other cases, the pipeline is a project containing programming code. An example is a Java project in which the pipeline has been "programmed," a feature of the Bamboo platform (Bamboo Java Specs). This results in a Java build performed by Maven or Gradle. The result is a pipeline, stored in an artifact repository. If the pipeline is generated using a Pipeline Generator tool, the input is the repository of the DevOps team. The Pipeline Generator tool "builds" the pipeline code after scanning the repository.
- *Perform unit tests*: The unit tests a pipeline of pipelines performs are nondestructive. All flows within a pipeline must be tested.[4] Variables must be overridden to mimic certain behavior. A destructive stage or task

[4] Unit tests and integration tests of pipelines can be combined.

must be "neutralized" by injecting code to make it a nondestructive stage/task or using a mock stage/task. The next chapter goes deeper into testing pipelines.

- *Analyze code*: Different types of pipeline code analysis are possible. Here's a summary:
    - The pipeline code must be valid; this can be done either as part of the entry criteria validation (preferred) or in the *Analyze code* stage.
        - In the case of a YAML file, the pipeline code must be valid YAML. Use yamllint, for example.
        - Validating a Jenkinsfile in Visual Studio Code (VS Code) can be done using the Jenkins Pipeline Linter Connector, but it is not very CI/CD-friendly. Jenkins itself also has a built-in linter that can be used to validate the Jenkinsfile in a pipeline.
- If the pipeline consists of code written in a programming language or script, regular code analysis tools can be used, such as SonarQube.
    - Pipelines can be analyzed on compliance. A compliance scanner validates whether company policies are applied to the pipeline code. Examples are as follows:
        - The pipeline must contain certain mandatory stages or tasks. Perhaps it is mandatory to include a SonarQube task in the pipeline or the pipeline must include a dual control stage before an artifact is allowed to be deployed to production.

    - It is not allowed to continue if errors occur in the pipeline. A pipeline must contain quality gates. If the conditions of the quality gate are not met, the pipeline should stop and return an error. If the quality gate is bypassed in the pipeline and the subsequent stages are still executed—including the stage in which the application is deployed to production—the pipeline violates a policy.
    - In the case of SaaS solutions, pipelines can run on generic SaaS nodes, agents, or containers. However, some organizations use a dedicated pool of nodes/agents/containers for security reasons. An organization policy may demand that pipelines are only allowed to run on a node/agent/container belonging to this pool. If the pipeline did not specify this pool, the compliance scanner marks this pipeline as noncompliant and cannot be used.

- *Package artifact*: If the artifact is the same as the original file, this stage has no purpose. If the input consists of a set of related files, it might be wise to pack all files into one `.zip` or `.tar` file, even if the *Execute build* stage is absent.
- *Publish artifact*: It makes sense to publish the pipeline code to a central repository, similar to what we do with application artifacts. Especially, if an external IT4IT team develops a base pipeline, it is convenient for DevOps teams to grab this base pipeline from a central repository.

- *Provision test environment*: The test environment of a pipeline is an ALM platform or integration server, which executes the pipeline tests. This is not the same environment in which a regular build, test, and deployment of the application take place. The pipeline test environment is a separate environment, specific to testing the pipeline. If, for example, the pipeline includes a task to tag the code in the SCM repository and this task is tested, it should not be done in the original repository. In a test environment, this task can just be executed, and no harm is done.

If possible, this pipeline test environment is an ephemeral environment that can be removed after use.

- *Deploy artifact to test*: The pipeline artifact is retrieved from the repository and deployed to the pipeline test environment. Additional files—needed by the pipeline to function properly—are part of this deployment, for example, a snapshot of the application code for which the pipeline was developed.
- *Perform test*: Several pipeline tests are performed. This means that the pipeline is executed in a pipeline test environment. These tests can be automated or manually executed. The following are things that make sense to test:
  - Validate whether the pipeline runs at all.
  - Validate whether input variables are defined and whether they contain the expected data type; e.g., validate whether a variable contains a numeric value or a date.

- Validate whether the correct application artifacts are built and deployed.
- Validate whether the pipeline flow works when running in different SCM branches.
- Validate whether directories, file locations, and files used in the pipeline can be reached and read.
- Validate whether all paths are executed in certain conditions.
- Validate whether external connections work.
- Validate whether the performance of the pipeline is sufficient.

Note In addition to a pipeline test environment, an application test environment is needed to deploy the artifact. This application test environment is either a fixed or ephemeral test environment.

- *Validate infrastructure compliance*: The production environment of the pipeline artifact is an ALM/integration platform. This is the same platform on which the application is built, tested, and deployed by the pipeline. Assuming that this platform already exists, the *Validate infrastructure compliance* stage and also the *Provision production environment* stage are not relevant.
- *Deploy artifact to production*: The stages *Validate exit criteria*, *Perform dual control*, and *Provision production environment* all seem a bit too formal for a pipeline artifact, and it can be assumed that the production environment is already present.

The question is also what it means to deploy the pipeline code to production. The formal route would be a "deployment" of the pipeline code to the production source code repository. Most likely, this source code repository is the same as the one that contains the application code and (the previous version of) the pipeline code. Creating a pull request—if used—and having it approved to merge the pipeline code in this repository seems to summarize the *Perform dual control* stage. Pushing the code to the original remote repository covers the *Deploy artifact to production* stage.

- *Notify actors*: Informing actors is still needed to keep them informed about the progress of the pipeline of pipelines.

Summarized, the pipeline of pipelines looks like a stripped-down version of the Generic CI/CD Pipeline.

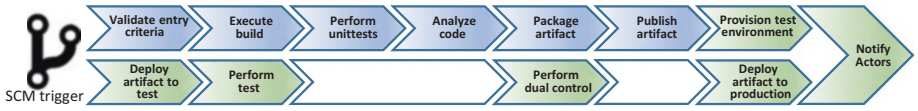


Figure 5-26. *Pipeline of pipelines*

Sustainable Pipeline Development

At the end of this chapter, it is important to highlight the environmental impact of pipelines. Sustainable computing is a relatively new topic, and people are not sure what measures they can take to limit the carbon dioxide footprint of their system. It is also impossible to come up with a complete list of recommendations on how to optimize pipelines, but here are some useful pointers:

- If you run an integration infrastructure, it is important to distribute the workload over smaller-sized (virtual) servers. Small servers with a higher utilization consume less energy than large servers that are underutilized. An underutilized server spends more time in an idle state, which consumes more energy.
- Consider running pipelines in the cloud. There is heavy pressure on cloud service providers (CSPs) to make their data centers more sustainable. The scale, decisiveness, and budget of CSPs go beyond the possibilities of a company's data center. Google, for example, has the option to choose a Google Cloud region according to the lowest carbon dioxide footprint. Microsoft claims that using the Microsoft Azure cloud platform can be up to 93 percent more energy-efficient and up to 98 percent more carbon dioxide efficient than on-premises solutions (see [30]). Amazon Web Services (AWS) is focused on powering its operations with 100 percent renewable energy by the year 2025 (see [31]).
- Sustainability of a SaaS ALM platform also depends on the (serverless) architecture of the platform itself. ALM platform developers could look into the possibility of implementing the tooling using functions/Lambdas.
- Choosing the right infrastructure definitely can have a positive impact, but also the use of certain scripts and the design of the pipelines can influence the carbon dioxide footprint. In the case of scripting, the language in which the script was written makes a difference. The energy consumption of Python, TypeScript, or JavaScript can be 60 times higher than languages

such as C, Rust, or even Java. In general—except for Java—one can state that compiled languages are more energy-efficient than interpreted languages [9]. Consider this during the realization of your pipelines.

- Even the pipeline design can be optimized to achieve a lower carbon dioxide footprint. Validating at the beginning of a pipeline run whether mandatory pipeline variables are defined and whether an external system can be reached prevents the pipeline from failing somewhere at the end, having consumed unnecessary energy.
- In the case of test tasks, some sustainability measures can be applied. The concept of *fail fast* implies that a pipeline stops as soon as a mandatory test task fails. If, for example, tests run in parallel and one of the mandatory tests fails, all parallel test tasks must stop immediately, as if someone presses a red stop button and the whole assembly line comes to a halt. There is no need to wait for the last test to finish.
- The *auto-cancel* option can be used if a new pipeline instance is started and the already running pipeline instance has become obsolete. The obsolete instance must stop to prevent burning unnecessary CPU cycles.
- Consider scheduling the *Analyze code* stage to be executed once a day. This introduces a slight risk, though. An artifact could be deployed to production before the scheduled *Analyze code* stage has run. There is a chance that this artifact contains a vulnerability. Accept the risk, and validate the *Analyze code* report afterward, as soon as it is available. If the number of

application code changes was significant, it is an option to manually trigger the *Analyze code* pipeline and validate the result before the deployment to production is approved.

- Consider moving the *Analyze code* stage just before the *Validate exit criteria* stage. The benefit is that no code is analyzed that did not pass the tests.
- If the team uses feature branches, do not analyze or test the code in a feature branch pipeline run, but only in the mainline run.
- Decompose the application into multiple independent components (microservices) and create a pipeline for each component. The benefit is that instead of building a big monolithic application after every code commit, only the components that are changed are built. This not only speeds up the pipeline execution time but also reduces compute cycles and saves energy.
- Optimize the use of test environments. If you have test environments that are not being used frequently or at all, they may still be consuming energy. To save energy and reduce waste, you can either power them down or remove them if they are no longer needed.
- One can question whether it is always needed to start a pipeline and execute all stages, even though the change in the code was very small. Here is where a rule-based trigger could step in. A rule-based trigger is a—still theoretical—trigger that decides when a pipeline starts. We are all familiar with SCM-event triggers and

scheduled triggers to start a pipeline. A rule-based trigger determines whether a pipeline starts based on certain rules. The following are examples of these rules:

- The pipeline does not start if code is committed with an associated work item with a *low* priority. Only code associated with work items with a *medium* or *high* priority results in the start of the pipeline.
- The pipeline starts after only x number of commits.
- The pipeline starts only after y percent of the codebase was changed.

It is unclear whether any tooling offers rule-based triggers out of the box at the time of writing.

Summary

You learned about the following topics in this chapter:

- There are three ways to create pipelines.
  - Using a user interface
  - Using a scripted pipeline
  - Using a declarative pipeline
- Pipeline specifications shift toward declarative pipelines, often in YAML notation.
- Modern platforms share some common features. Integrating them in pipelines as a pipeline language construct reduces complexity.

- External libraries, environmental properties, and secrets in pipelines are explained.
- Security issues concerning external libraries, environmental properties, and secrets are highlighted. Solutions are presented on how these risks can be mitigated.
- There are several CI/CD-related development areas at different places within an organization, with each area covered by a specific type of team (SaaS provider, IT4IT team, or DevOps team).
- There are different approaches toward pipeline development, each with its pros and cons.
    - Simplified development
    - Extended development
    - Advanced development
    - Developing base pipelines
    - Pipeline generation
- The concept of pipeline of pipelines was explained.
- Tips were given to develop sustainable pipelines.

CHAPTER 6

Testing Pipelines

This chapter covers the following:

- The importance of testing pipelines.
- How to create a unit test using a test framework. The chapter describes how the pipeline is manipulated by the test framework and executed in a pipeline test environment.
- An example of a pipeline performance test and how overall execution time is improved by the parallelization of activities.
- The concept of pipeline acceptance testing in simplified and advanced pipeline development.

Testing Pipelines

Pipelines and testing can be highlighted from different viewpoints. Most books and articles describe how pipelines are used to test an application, which test frameworks are used, and how everything integrates into the pipeline. Chapter 4 highlights the importance of a test strategy and how this reflects on the pipeline design.

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_6

What is often neglected, but equally important and interesting, is testing the pipelines themselves. This chapter is dedicated to pipeline testing.

Testability of Pipelines

Pipelines are defined as code. Code can be tested. Most declarative pipeline code (with some exceptions) consists of YAML files or scripts. Testing them is a challenge. Teams often test the pipelines using trial and error, sometimes screwing things up because a wrong version of an app was deployed by accident. In some cases, code from a feature branch was accidentally tagged with a release version tag, and because of the trial-and-error nature of developing and testing pipelines, the number of commits is very high. The once well-organized overview with regular application pipeline runs is cluttered with a zillion test runs. Testing pipelines is hard because teams also don't have the tools to test properly.

Just as with testing applications, pipeline code must be tested in a test environment. The pipeline test environment must differ from the environment in which the business application is built, tested, and deployed. From a pipeline point of view, the environment used to build, test, and deploy the business application is considered the production environment. The pipeline test environment is either a separate ALM platform or integration server infrastructure or an infrastructure in which separation between the regular pipeline environment and the pipeline test environment is established in another way. Important is that the pipeline must be able to run in a test/sandbox environment, without the destructive character. It must also be possible to test specific characteristics of the pipeline. This means the following:

- Checking the configuration of the pipeline and its components to ensure that they are set up properly and functioning as expected. This can include things like

verifying that the correct tools and dependencies are being used and that variables are configured.

- Pipeline unit tests are focused on testing individual parts of the pipeline. Pipeline unit tests are performed on a local development machine (if possible) and also in the pipeline of pipelines (if used).
- All flows within the pipeline are tested by simulating a real-world deployment scenario and checking that all components of the pipeline work together properly. This includes an end-to-end test, in which the entire pipeline is tested, from code commit to deployment, to ensure that it works as expected in a real-world scenario.
- Quality gates must be tested; does the pipeline break if certain quality criteria are not met?
- The performance of the pipeline must be tested to detect queuing or potential bottlenecks in execution speed.
- The pipeline code must be analyzed for quality, security, and compliance; does it adhere to organizational policies?
- Pipeline tests must be able to run in a "sandbox" or test environment to prevent destructive actions, for example, to test tagging of a commit in the repository with a release version, without actually tagging it in the original repository in which the pipeline and application code is stored.

To properly test a pipeline, a few test types must be performed.

- Unit (and integration) tests
- Performance tests
- Pipeline compliance and security tests
- Acceptance tests

Let's discuss them in the next few sections and point out how this can be done.

Unit Tests

Let's face it. Test frameworks for pipelines are almost nonexistent or at least very scarce. Also, when dealing with SaaS platforms of big-tech companies, you might expect that there is some information or support concerning pipeline testing. The platforms are mature, but testing pipelines are not given that much TLC. Local testing of pipelines within an IDE is very much desired but often not supported. Mocking a task, so it is not really executed, is a simple feature, but which provider supports this?

Sometimes the only thing left is to develop something yourself. As an example, unit testing an Azure DevOps pipeline is explained in this section. This is a real example using a relatively simple unit test framework.[1]

The example makes use of a build unit test framework to manipulate the pipeline and communicate with the Azure DevOps platform. The framework makes use of JUnit 5 as a testing framework and uses the snakeyaml and jgit libraries.

---

[1] The code of this framework is published on the Github page of the author, however it is still experimental at this stage.

To manipulate the pipeline and make it testable, the unit test framework implements the following features:

- The pipeline, which consists of one or more YAML files, is wrapped into a Java (pipeline) object and loaded into the Junit test class. The Junit tests make use of various methods that help in realizing different test cases.
    - It must be possible to override variables and parameters in the pipeline.
    - It must be possible to skip (disable) certain pipeline stages, jobs, and steps.
    - It must be possible to add a clause to continue in case of an error.
    - It must be possible to inject custom code into the pipeline.
    - It must be possible to stub/mock tasks in the pipeline. This means it must also be possible to mock a deployment, for example. This can be realized by replacing a task with a script task with some custom code.
    - If mocking is not used for some reason, it must be possible to intercept commands to prevent disruptive actions (using a dry-run flag, for example).
    - It must be possible to mimic other branches, replacing the current branch with a given branch.
    - It must be possible to check on fail-fast behavior; if a unit test fails, the pipeline must stop immediately (after notification).
    - It must be possible to retrieve the results of the pipeline run.

- The manipulated pipeline code (a modified copy of the original pipeline code) is deployed to an Azure DevOps test environment (a specific Azure DevOps project used for testing pipelines) from where it is started.
- The `pom.xml` file is updated; the `connection` and `developerConnection` in the `pom.xml` file must point to the Azure DevOps test environment (project) instead of the original Azure DevOps project.
- Pipeline results are retrieved using an Azure DevOps API and exposed in JUnit tests. They are used to check whether the outcome matches the prediction. Unfortunately, the Azure DevOps API provides only rudimentary test results.

Figure 6-1 visualizes the setup.

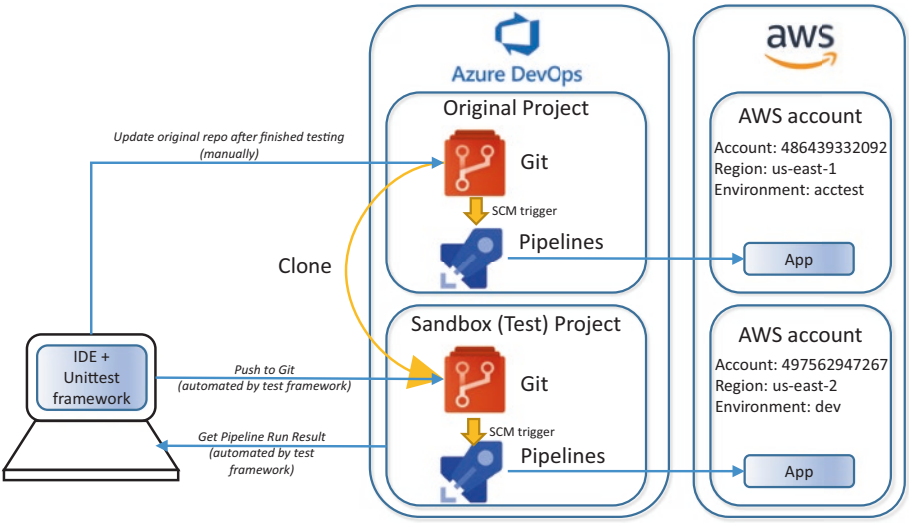


Figure 6-1. *Setup unit test Azure DevOps pipelines*

With this picture in mind, consider the following steps. The application code is located in a Git repository in the original Azure DevOps project. This code is cloned to another repository in an Azure DevOps test project. This test repository is checked out (manually) and resides on the workstation of the developer.

The developer starts developing a pipeline, as listed in Listing 6-1. This is the YAML file with the name `pipeline.yml`. For readability reasons, various stages are omitted from this pipeline.

Listing 6-1. pipeline.yml

```
name: $(Date:yyyyMMdd)$(Rev:.r)

parameters:
- name: environment
  type: string
  default: acctest
  values:
  - dev
  - systest
  - acctest
  - prod

variables:
- name: aws_connection
  value: 486439332092
- name: aws_region
  value: us-east-1

stages:
- stage: Execute_build
  displayName: Execute build
  condition: always()
  jobs:
```

```
  - job: Tasks
    pool: Default
    steps:
    - script: echo 'Execute build'
    - task: Maven@3
      displayName: Maven Package
      inputs:
        mavenPomFile: pom.xml
      condition: always()
    - task: CopyFiles@2
      displayName: Copy Files to artifact staging directory
      inputs:
        SourceFolder: $(System.DefaultWorkingDirectory)
        Contents: '**/target/*.?(war|jar)'
        TargetFolder: $(Build.ArtifactStagingDirectory)
    - upload: $(Build.ArtifactStagingDirectory)
      artifact: drop

- stage: Analyze_code
  displayName: Analyze code
  condition: eq(variables['Build.SourceBranchName'], 'main')
  jobs:
  - job: Tasks
    pool: Default
    steps:
    - script: |
        pip install whispers
        whispers ./

- stage: Deploy_artifact_to_test
  displayName: Deploy artifact to test
  condition: eq(variables['Build.SourceBranchName'], 'main')
```

```
jobs:
- deployment: Deploy
  pool: Default
  environment: ${{ parameters.environment }}
  strategy:
    runOnce:
      deploy:
        steps:
        - task: AWSShellScript@1
          inputs:
            awsCredentials: $(aws_connection)
            regionName: $(aws_region)
            scriptType: inline
            inlineScript: |
              #!/bin/bash
              set -ex
              export artifact=`find $(Pipeline.Workspace)/. -name
              'cdk*.jar'`

              echo "Deploying stack"
              cdk deploy --app '${JAVA_HOME_11_X64}/bin/java -cp
              $artifact com.myorg.myapp.Stack' \
                  -c env=${{ parameters.environment }} \
                  --all \
                  --ci \
                  --require-approval never
          displayName: Deploy to AWS
```

This pipeline builds a Java artifact (application) using Maven, after which a security scan is performed using the tool Whispers. This scan is performed only in case the branch in which the pipeline resides is the main branch. If the current branch is the main branch, the artifact is

deployed to an existing AWS account with account ID 486439332092 in a certain region (us-east-1; N. Virginia). Within each AWS account, virtual test environments are created, and the artifact runs in one of these virtual test environments. By default, the virtual test environment is the acceptance test environment (acctest).

When the pipeline is (unit) tested, a couple of actions are performed. The developer creates the unit tests, commits them, and runs the unit tests. This invokes the unit test framework, which makes a copy of the `pipeline.yml` files, and manipulates it according to the JUnit test. The manipulated pipeline file is then pushed to the test repository, and the pipeline in the Azure Test project starts running. The results of the pipeline run are retrieved—using an Azure DevOps API call—after each test is finished.

To make manipulation of the pipeline possible, the JUnit tests are defined as shown in Listing 6-2.

Listing 6-2. PipelineUnit.java

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import java.io.IOException;

public class PipelineUnit {

    private static AzDoPipeline pipeline;

    @BeforeAll
    public static void setUpClass() {
        System.out.println("setUpClass");

        // Initialize the pipeline
        pipeline = new AzDoPipeline("pipeline.yml");
    }
```

```
@Test
public void test1() {
    // Validate the pipeline flow in case the current
    branch is a feature branch (instead of the main branch)
    System.out.println("\nPerform unit test: test.test1");
    pipeline.overrideCurrentBranch("myFeature");
    try {
        pipeline.startPipeline();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    assertEquals (RunResult.succeeded, pipeline.
    getRunResult());
}

@Test
public void test2() {
    // Test the build and deploy stages:
    // - Use a different AWS account (Ohio based) for
    deployment
    // - Use a different environment (dev instead of
    acctest) for deployment
    // - Skip the 'Analyze code' stage, only the deployment
    needs to be tested
    System.out.println("\nPerform unit test: test.test2");
    pipeline.overrideVariable("aws_connection", "
    497562947267");
    pipeline.overrideVariable("aws_region", "us-east-2");
    pipeline.overrideDefaultParameter("environment", "dev");
    pipeline.skipStage("Analyze_code");
```

```
        try {
            pipeline.startPipeline();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        assertEquals (RunResult.succeeded, pipeline.
        getRunResult());
    }

    @AfterAll
    public static void tearDown() {
        System.out.println("\ntearDown");
    }
}
```

Unit test number 1 (`test1`) mimics the current branch. What happens in `test1` is that the current branch is replaced with `myFeature`, so the pipeline behaves as if it resides in the branch `myFeature`, even if it resides in another branch.

The pipeline code in unit test number 2 (`test2`) is changed by the unit test framework in such a way that deployment of the application artifact to AWS does not impact the current application in AWS. In `test2` the AWS account variables are replaced by other values, and the *Analyze code* stage is set to skip. This results in a unit test that is performed in a different AWS account, with account ID 497562947267. The application is even deployed in a different region (us-east-2; Ohio) and a different virtual environment (dev). To speed up the test, the *Analyze code* stage is skipped.

The pipeline, manipulated as part of JUnit `test2`, results in the code in Listing 6-3.

Listing 6-3. Manipulated Version of pipeline.yml as a Result of JUnit test2

```
name: $(Date:yyyyMMdd)$(Rev:.r)

parameters:
- name: environment
  type: string
  default: dev
  values:
  - dev
  - systest
  - acctest
  - prod

variables:
- name: aws_connection
  value: 497562947267
- name: aws_region
  value: us-east-2

stages:
- stage: Execute_build
  displayName: Execute build
  condition: always()
  jobs:
  - job: Tasks
    pool: Default
    steps:
    - script: echo 'Execute build'
    - task: Maven@3
      displayName: Maven Package
      inputs:
```

```
        mavenPomFile: pom.xml
      condition: always()
    - task: CopyFiles@2
      displayName: Copy Files to artifact staging directory
      inputs:
        SourceFolder: $(System.DefaultWorkingDirectory)
        Contents: '**/target/*.?(war|jar)'
        TargetFolder: $(Build.ArtifactStagingDirectory)
    - upload: $(Build.ArtifactStagingDirectory)
      artifact: drop

- stage: Analyze_code
  displayName: Analyze code
  condition: eq(true, false)
  jobs:
  - job: Tasks
    pool: Default
    steps:
    - script: |
        pip install whispers
        whispers ./

- stage: Deploy_artifact_to_test
  displayName: Deploy artifact to test
  condition: eq(variables['Build.SourceBranchName'], 'main')
  jobs:
  - deployment: Deploy
    pool: Default
    environment: ${{ parameters.environment }}
    strategy:
      runOnce:
        deploy:
```

```
steps:
- task: AWSShellScript@1
  inputs:
    awsCredentials: $(aws_connection)
    regionName: $(aws_region)
    scriptType: inline
    inlineScript: |
      #!/bin/bash
      set -ex
     export artifact=`find $(Pipeline.Workspace)/. -name
     'cdk*.jar'`

      echo "Deploying stack"
      cdk deploy --app '${JAVA_HOME_11_X64}/bin/
      java -cp $artifact com.myorg.myapp.Stack' \
          -c env=${{ parameters.environment }} \
          --all \
          --ci \
          --require-approval never
  displayName: Deploy to AWS
```

Note Azure DevOps does not support disabling stages at the moment. To skip a stage, a condition is used.

This way of testing has a lot of advantages. Without constantly changing the original YAML file and committing it in an SCM, the pipeline is manipulated by the JUnit test cases instead. This test approach is simple and also prevents the following:

- High commit rates in the original SCM repository because the original YAML file is not changed constantly.

- Errors slipping in as a result of constantly changing the YAML file.
- Pollution of the SCM history, pipeline dashboards, and pipeline overviews. Because the tests run in another Azure DevOps project, the SCM history, the pipeline dashboards, and the pipeline overviews of the original Azure DevOps project are not affected.
- Long wait times. If you want to focus on the test of a certain stage or task, it is easy to skip stages or tasks you don't want to see run. This only costs time.
- Other destructive actions, such as tagging the application code in the code repository, tagging the pipeline, or deploying the application to a test environment, which is already in use by the QA team.

Performance Tests

Performance testing does not apply to the performance tests of the application, but to the performance test of the pipeline itself. Important to keep in mind is that fast feedback is of utmost importance. The processing time of the pipeline must be as short as possible. Your pipeline may be affected by various types of performance penalties.

- The execution time of the pipeline takes too long. One underlying problem could be that compute and/or storage capacity is insufficient. This can be solved by scaling up the infrastructure.
- Another reason why the execution time of a pipeline takes too long is that the design is not optimized for speed. The solution can be found in revising the build strategy and/or redesigning parts of the pipeline.

- Queuing occurs. The time a pipeline stays in the queue adds up to the processing time of the pipeline. Scaling up the infrastructure and applying fine-grained prioritization are solutions that could solve this.

Performance tests are focused on both pipeline execution time and pipeline queuing time. It has been discussed how parallelism helps in speeding up the pipeline processing time. Let's put it to the test and look at a real example of a pipeline containing *Execute build*, *Analyze code*, and *Deploy artifact to test* stages.[2] The *Execute build* stage contains a Maven build task, building a Java application.[3] The *Analyze code* stage contains three tasks: a SonarQube scan, a Fortify scan, and a Whispers scan. The *Deploy artifact to test* stage deploys the artifact to an AWS test account. All stages and tasks are executed in sequential order, resulting in the execution times shown in Figure 6-2.

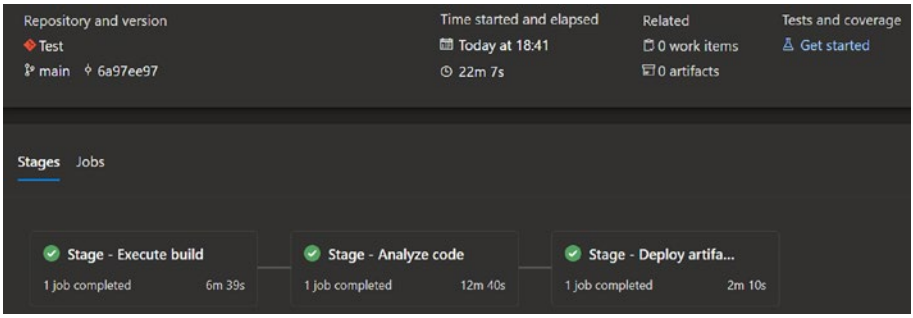


Figure 6-2. *Stages in sequential order*

[2] Queuing time was not measured, so this was not taken into account.

[3] This application has a relatively large codebase, so the effect of parallelization becomes apparent.

Most platforms provide the necessary information related to the performance of a pipeline. The processing times of stages, jobs, tasks, and the pipeline as a whole are shown in Figure 6-2. The figure shows that the overall execution time of this pipeline is 22 minutes and 7 seconds. This is not very fast. The *Analyze code* stage costs relatively a lot of time and contributes a lot to the overall processing time. The *Analyze code* stage is a bottleneck. The individual tasks in the *Analyze code* stage are set up in such a way that they do not depend on the output of the *Execute build* stage.

So, nothing prevents us from executing the *Analyze code* stage in parallel instead of sequentially executing the three stages. This considerably shortens the overall execution time of the pipeline. This is clearly expressed in the pipeline run in Figure 6-3. The *Execute build* and *Analyze code* stages run in parallel. The *Deploy artifact to test* stage has a dependency on the *Execute build* stage and can be started only after the artifact has been built. There is little that can be optimized here. Running the *Analyze code* stage in parallel reduces the total execution time of the pipeline to 12 minutes and 43 seconds. Note that the *Deploy artifact to test* stage does not wait until the *Analyze code* stage is finished.

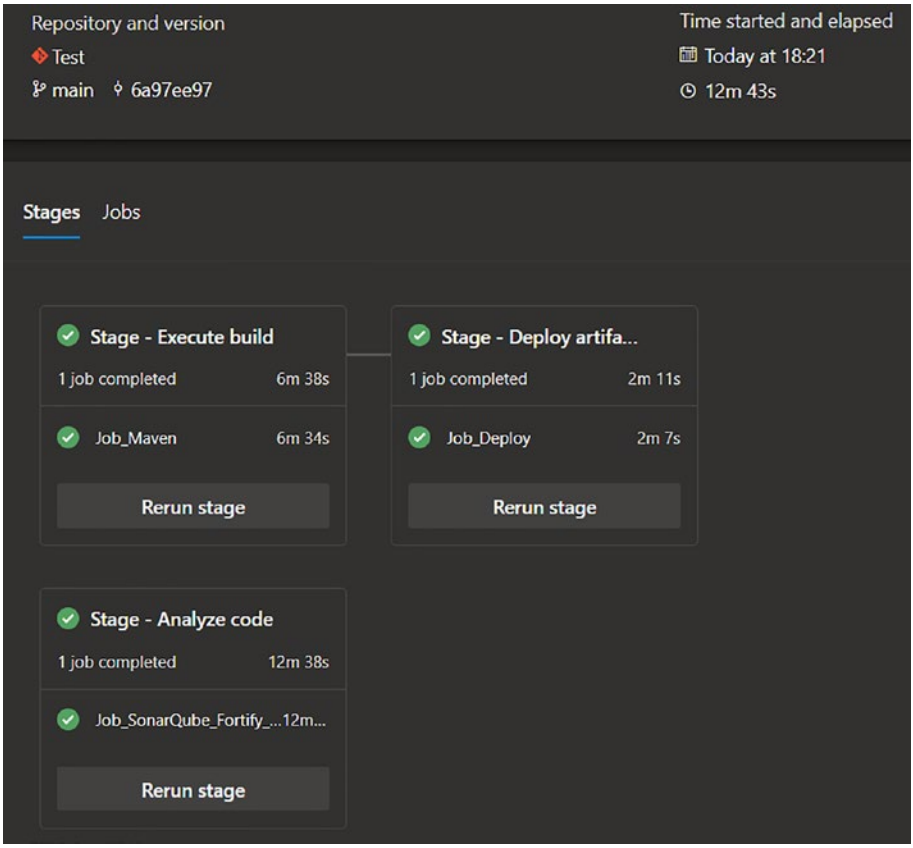


Figure 6-3. *Analyze code stage in parallel*

Of course, it is possible to optimize this a bit more. The individual tasks in the *Analyze code* stage are still executed sequentially. These can also be run in parallel. Let's see what that brings us; see Figure 6-4.

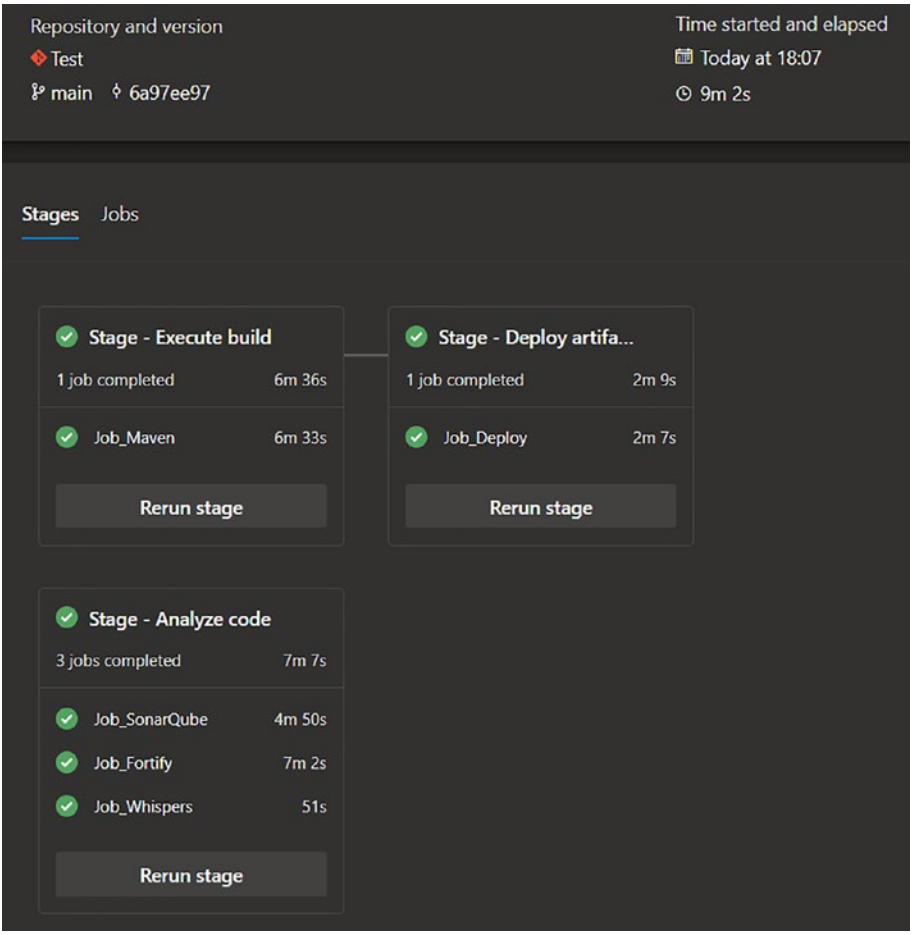


Figure 6-4. *Analyze code tasks in parallel*

Again, the total pipeline execution time has been brought back to normal proportions, and the pipeline fully executes well within 10 minutes. Of course, this is just one of the measures to increase pipeline performance. Experience showed that after applying a combination of measures such as *pipeline caching*, *parallelization*, and *multithreaded builds*, pipeline execution time could be reduced by 75 percent.

As shown in this example, the platform registers the execution times of multiple levels in the pipeline, and this information can be used to spot bottlenecks in the pipeline execution.

Pipeline Compliance and Security Tests

A pipeline must be checked for potential vulnerabilities to ensure that it is configured properly to protect sensitive data and prevent unauthorized access. This can include things such as checking for vulnerable dependencies with external tools, testing authentication, checking access controls, and conducting regular security audits.

In addition, some business organizations define policies to which a pipeline must comply. This means certain settings are prohibited or certain tasks are mandatory. If the pipeline does not meet these policies, it is blocked from execution or reported to a central department. This compliance check is not performed by the pipeline itself, of course; that would not make any sense. The compliance check is integrated into the platform, as a hook or decorator, for example. This adds a pre-job to the pipeline, which is always executed as the first action before the pipeline starts. If the pipeline does not comply, the execution is aborted or the noncompliant pipeline is reported. Here are some examples of policies that organizations could enforce:

- One of the policies an organization may enforce is the existence of certain stages and/or tasks. For example, the pipeline must have an *Analyze code* stage with two mandatory tasks that perform a SonarQube, a NexusIQ, and a Whispers scan.
- A deployment to production must have a *Perform dual control* stage. Pipelines without this stage are

not compliant. In addition, if the pipeline was started manually, it may not be approved by the same person.

- Some platforms have the opportunity to continue if a certain error occurs. This could also mean that some quality gates can be bypassed. Tasks with the setting `continue on error = true` are reported or blocked.
- Stages or tasks executed on a nonproduction server/node are reported or blocked. On some platforms, it is possible to assign your laptop as a server to execute a pipeline. Consider the risk if the deployment of an artifact to production is executed on a nonsecured laptop.
- To enforce the requirement "Do not retrieve libraries or external resources directly from an Internet location," the pipeline is scanned for service connections with a nonauthenticated endpoint. If the pipeline uses such a service connection, it is reported or blocked.
- Artifacts must be stored in a binary repository. One of the policies may enforce the existence of a *Publish artifact* stage using a service connection with a specific endpoint configured.
- All resources that contribute to the creation and deployment of a release to a production environment must be prohibited from deletion. This applies to code repositories, pipelines, and artifacts.

Acceptance Tests

Whether the development team uses simplified pipeline development or advanced pipeline development, at some point the pipeline must be accepted for usage.

Validating the quality of the pipeline in simplified pipeline development poses risks because the pipeline is not thoroughly tested. Acceptance tests do not play an explicit role in simplified pipeline development. Accepting the quality of the pipeline is implicit. It is a process of changing the pipeline, pushing it to a repo, and watching its behavior. If it does not work properly, this step is repeated. Accepting the pipeline is nothing more than continuously implementing the adjusted pipeline and seeing it working in its normal environment until the expectations are met.

An acceptance test in advanced pipeline development involves the execution of all the stages in the assembly line. This includes a *Perform test* stage in which the pipeline is executed in a pipeline test environment. If all stages in the assembly line are passed, the quality of the pipeline can be considered sufficient, and the pipeline can be implemented (used).

Summary

You learned about the following topics in this chapter:

- Unit testing can be performed using a separate pipeline test environment; unit testing was demonstrated using an example in which a unit test framework was used.
- Pipeline testing in a separate test environment prevents high commit rates, destructive actions, pollution of the SCM history, and pollution of the dashboards and overviews of the regular pipeline environment.

- Executing pipeline performance tests can help with spotting bottlenecks in execution speed.
- Pipeline compliance checks can be used to improve pipeline quality and to meet organization policies regarding pipelines.
- Pipeline acceptance tests are more explicit if the pipeline development quality improves.

CHAPTER 7

Pipeline Implementation

This chapter covers the following:

- What pipeline implementation involves
- The organizational impact if a new or updated pipeline is implemented and used
- The different types of operating models concerning integration infrastructure and the responsibilities of the team and the organization
- How an application implementation can benefit from using additional features such as a runbook, a release note, and artifact promotion

When an application is deployed to production for the first time, a lot of things have to be arranged. Certificates must be requested and installed. Credentials and other secrets must be arranged and stored safely. Application monitoring must be set up, etc. Assuming that not all activities in the software supply chain are, or can be, automated, some manual tasks are involved. In addition, the process of managing and using the application must be in order. The team must know what to do if an application fails or behaves badly. Procedures for change, incident, problem, and availability management must be in place.

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_7

The same applies to a pipeline. The environment in which the pipeline runs has to be prepared and scaled. The platform on which the pipeline runs may have external connections that need to be secured, and both the integration platform infrastructure and the pipelines themselves need to be monitored. If a pipeline fails or does not work the way it should, the team must react properly. Only after these preparations have been done is the pipeline ready to be used. Implementing a pipeline involves more work than meets the eye.

Pipeline Implementation

The implementation of a pipeline itself is a bit odd. If the implementation of a pipeline is compared with the implementation of an application, the pipeline needs to be configured for and deployed to a target environment. But what is the target environment in the case of pipelines, and can we speak of the deployment of a pipeline? In the pipeline of pipelines discussion, the conclusion was that deploying a pipeline to production is nothing more than pushing the pipeline code to the remote repository and merging it with the mainline. Figure 7-1 illustrates this behavior.

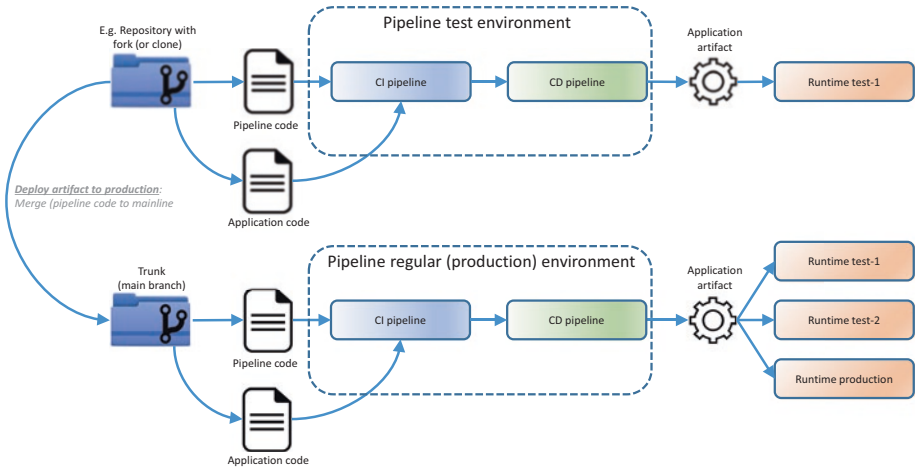


Figure 7-1. *Pipeline implementation*

In Figure 7-1, the new or updated pipeline code resides in another branch in the same repository as the application, or it resides in another repository, which is cloned from the original repository. This depends on the pipeline development method. The pipeline code is tested in a pipeline test environment, which builds and deploys the application to a test environment, called *Runtime test-1*. Merging the pipeline code with the mainline means that the pipeline from that moment is implemented and can be used to deploy to all target environments (*Runtime test-1*, *Runtime test-2*, and *Runtime production*).

Organizational Impact

A pipeline is developed according to requirements and guidelines and properly tested before it can be used. This means the functional behavior is according to the specifications, the performance of the pipeline is tested and meets the criteria, security measures are in place, and the pipeline meets the compliance specifications of the organization. Because the pipeline is used by the DevOps team, all team members must be confident

that it is usable. If needed, documented instructions about the pipeline's use, technical setup, and maintenance are drafted. This is not mandatory, but it can help to get the whole team prepared. The team decides whether documentation is needed. They must approve the readiness of the pipeline.

Every time a new version of a pipeline is used, its limitations must be recognized, and known issues should be logged. Register the requirements that were not realized. Register mitigating actions, such as manual checks, if some requirement is not implemented but may pose a risk.

Both the team and the business evaluate possible gaps and other improvements that can be made. Each gap is put on the backlog and is prioritized. The involvement of the business is mainly about the release strategy and the use of organization policies. Is it needed to deploy each realized feature to production within 15 minutes, or is it still sufficient to combine features into small increments and release them with a one-week frequency? If something changes in these aspects, the workflow of the team and possibly the design of the pipeline are impacted. A governance structure has to be in place to perform these evaluations. Make the time as a team to evaluate or, even better, establish a process of continuous improvement, not only for application development but also for pipeline development.

If not configured already, define what the notification structure should look like. During the development of the pipeline, the whole team probably receives the same email with the status of the pipeline run and with the request to approve a deployment. Just before implementing a pipeline, recipients must be configured, and notifications are assigned, so every team member receives only the specific information in which they are interested. Prevent information overloading and make use of dashboards to visualize important information.

Team Discipline

Even if the team is enthusiastic about automation and working with pipelines, it still happens that certain things are a bit neglected. Pipeline implementation also means that the team must be disciplined in certain areas. Some persistent problems are the following:

- *Breaking builds*: One of the principles of continuous integration is that broken builds must be repaired immediately. Developers are expected to drop what they are working on and solve the broken pipeline. This is a bit of wishful thinking. Developers often don't react immediately to this event. That doesn't have to be a problem if it doesn't lead to issues with releasing an artifact too late. However, leaving the pipeline broken for one or two days is also not a recommended practice. One obvious reason why a pipeline can break is that the committed code is incorrect and cannot be built. Another reason is that the world around the pipelines is in flux. External systems can be down, updated, or not accessible anymore; vulnerability checks are tightened; credentials or certificates are expired; or the ALM/integration platform itself suffers from technical problems. Teams must repair these broken pipelines. Otherwise, the effort to repair them will increase as time goes on.
- *Disabled quality gates*: Good practice is that if the code analysis detects severe or high-ranked vulnerabilities in the code, the pipeline "breaks" because the quality gate kicks in. Some pipelines do not have this quality gate activated, either by accident or on purpose. The latter is probably because of the following issue.

- *Follow-up on code analysis defects*: Some teams have good code analysis hygiene. They solve all the important vulnerabilities, so the quality gate is passed. Other teams neglect the code analysis results, disable the quality gate, and build up technical debt.
- *Unit test coverage*: The same applies to unit test coverage. Some teams make it a sport to keep the coverage high. Other teams do not do so. Low unit test coverage—coverage below a predefined threshold—should break the pipeline.
- *Automating tests is lagging*: There is often a backlog in automating tests. Sometimes, the number of people involved in automating tests is limited, causing a large backlog in test automation. This can happen if test automation is performed solely by a (small) QA team. It helps when test automation activities are spread over the team and developers are also involved in developing automated tests.

Depending on the team and its maturity, there are more persistent problems. Some teams still manage to bypass the pipeline and deploy to production in another way, or they perform continuous integration of a develop branch in the pipeline, while still creating a release artifact from the main branch on their local development machine. If the pressure is on, pull requests are approved without looking at the code. This is all part of growing up, but these problems must be addressed.

Integration Platform

Depending on the type of integration platform used, the responsibilities of setting up and managing the infrastructure differ. In this context,

integration infrastructure involves the integration platform (middleware) like Jenkins and additional tooling used by pipelines such as SonarQube, deployment tools, etc. Integration infrastructure also includes the infrastructure on which this all runs. This results in several operating models.

SaaS model: A complete SaaS solution offers a full integration platform, creating fewer concerns for the DevOps team. Because of the shared responsibilities, the DevOps team can solely focus on implementing pipelines, while the SaaS provider is responsible for managing the complete integration platform stack, including hardening and scaling servers and regularly patching the software. Platforms such as Azure DevOps or CircleCI Cloud fall into this category.

IaaS model: It is also possible to make use of infrastructure as a service (IaaS) in which the infrastructure provider manages the server landscape, while the integration platform stack is managed by either a separate IT4IT team or the DevOps team itself. In this context, the IT4IT, or DevOps, team gets more responsibilities, from managing a Kubernetes cluster to regularly upgrading containers, patching software, and installing plugins. The team also has to determine whether the platform is sufficiently scaled. Maybe the infrastructure was set up once, but pipeline performance tests showed that the capacity is not enough anymore with the introduction of new pipelines. Rescaling the infrastructure is required, so the performance criteria are met again. In

addition to this, offloading work to separate servers/containers must be considered. If the build, code analysis, tests, and deployment are all executed on the same server, moving certain stages to other servers/nodes/agents helps with spreading the load. This type of operating model can typically be achieved with Jenkins installed on plain servers such as AWS EC2 and Azure VMs or Jenkins in a Docker container running on Azure Kubernetes Service (AKS) or even AWS ECS Fargate.

Self-hosting model: An organization can also decide that it wants to host the complete integration platform infrastructure. This means that even more preparations are needed. The following are additional responsibilities:

- Provision the infrastructure on which the integration platform runs.
- Logging, monitoring, and alerting of the infrastructure must be set up and configured. Determine which system metrics need to be validated; for example, an alert is raised if a server uses more than 90 percent CPU capacity or an alert is raised if disk space is greater than 80 percent.
- The final infrastructure needs to be approved. Use a reference framework such as ISO 25010 (see [25]) as a guideline to determine whether all (nonfunctional) requirements are fulfilled, and make sure that the infrastructure is secure enough. In the latter case, a reference framework like the NIST Framework for Improving Critical Infrastructure Cybersecurity can be used (see [26]).

In addition to the already mentioned infrastructure preparations in the various operating models, security measures need to be applied to the infrastructure. Here are a few examples:

- Is the infrastructure secure enough? Make sure all servers are hardened and vulnerability management is in place. Patch the servers regularly.
- Are all connections secure? The ALM platform/integration platform maybe communicates with an SCM system, a work item management system, servers performing code analysis, etc. Connections need to be secured using HTTPS, for example (and preferably using mTLS instead of single-sided TLS).
- This also applies to connections with target environments—both test and production—on which the application runs.
- Refine access by setting permission for a user or a group. Users who manually start a deployment are not allowed to approve the deployment themselves.
- Configure branch policies if not done already. If the team uses branches and the pipeline associated with a branch fails, it should not be possible to merge that branch into the mainline.
- Configure a vault used to store tokens, keys, credentials, and other secrets.
- Configure the infrastructure in such a way that application code, pipeline (runs), test runs, work items, pull requests, etc., involved in the creation of a release artifact, which is deployed to production, cannot be deleted.

- Install a pipeline compliance scanner. The scanner validates whether pipelines comply with company policies.

Target Environment Preparations

If the team has adopted *extended* (or *advanced*) pipeline development, most of the development and pipeline tests were executed using a pipeline development/test environment. If the pipeline is ready to be implemented and used, it is promoted, so it can build and deploy the application to various target environments. This may include additional test environments and a production environment. Configure these target environments to be accessible to the pipeline and deploy over a secure connection.

The deployed application probably also needs (database) credentials, certificates, or static data. This has to be requested or generated and propagated to the target environment so the application can use it. Preferably, this is an automated process; use operational pipelines to establish this. The next chapter deals with operational pipelines in more detail.

In the case of an application test environment, test data needs to be arranged. Either generate synthetic data or use a copy from production, but make sure to anonymize the data.

Playbook

What is the business impact if an incident or a problem with a pipeline occurs? A failure of a pipeline may lead to damage. For example, an urgent application fix is created and needs to be deployed. However, the pipeline does not work because of an infrastructure failure of the integration platform. This could damage the continuity of a business process if

the pipeline is unavailable for a long time. ITIL processes also apply to pipelines. Playbooks can play a useful role in incident and problem management processes.

A playbook contains documented investigation methods to detect and resolve problems. They are useful for investigating incidents or failures. Playbooks can also be used for pipelines. Drafting pipeline playbooks can already be started during pipeline testing. Common pipeline failures and solutions are added to the playbook. Of course, playbooks are never complete, and after implementation and usage of the pipelines, more cases will occur. These cases are also added to the playbook.

Application Implementation

It is hard to speak about pipeline implementation without mentioning application implementation. Application implementation is, after all, the goal of using a pipeline in the first place. Adding certain features to a pipeline can contribute to a solid application implementation experience. Consider using or implementing these features.

Runbook

> *"A runbook is a set of processes and procedures that you execute repetitively to support various enterprise tasks."*
>
> Reference [33]

Why do you need a runbook if you use automated pipelines? That is a good question. A pipeline is already orchestrating the implementation of an application, right? But teams do still work with a runbook even if they also make use of pipelines. There are a couple of reasons why the use of a runbook is still valid.

- There are still one-off tasks or activities that are not part of CI/CD. The start of CI is a commit to a repository. The end of CD is the deployment of an artifact to a production environment. Plenty of tasks fall into the processes before and after CI/CD. Think about requesting an Azure subscription, configuring the IAM roles, and assigning team members. In addition, regular maintenance or migration involves activities that are also not part of a CI/CD pipeline. Sometimes these activities are complex and require a detailed runbook.
- Another reason to use a runbook is the first-time implementation of a complete system. You don't have CI/CD arranged on day one. The implementation of a new system in production maybe requires the execution of several pipelines in a specific order; even in the case of a microservice architecture, some pipelines need to run in a specific order. Think about setting up the base infrastructure components used by all microservices.

Everything can be automated, even a runbook. So, if a simple spreadsheet is not sufficient, use one of the several automated runbook tools. And because you already developing pipelines, setting up an orchestration pipeline to implement the runbook is also an option. However, the question is whether the benefit outweighs the effort and money spent. That is a question only the team can answer.

Release Note

A release note is a change log, describing the updates of the software. It may also include proof that all new features are tested and accepted. So, a

release note is associated with an artifact and contains information about the delivered features and (optionally) a test report. Because this book is about CI/CD design, the creation of release notes should not be done by hand but created automatically. There is one thing to consider, though. Between two production releases, there are probably multiple release candidates, including new features and changes. The last release candidate is marked as "the release" and deployed to production. Potentially, multiple release notes are created in between, each one associated with a release candidate. Only the final artifact deployed to production consists of all new features since the previous production deployment. Most likely the latest release note is very concise, describing just a bug fix. This is a bit unfortunate. The release note of the production artifact ideally consists of all changes between the previous production release and the current production release. In addition, the release note should also contain all test results performed on the release deployed to production.

To solve this problem, the system must keep track of all changes between the latest and next production releases and assemble all intermediate metadata to form an aggregated release note. After each production deployment, the status of the metadata is reset, and the assemble process restarts again. See Figure 7-2.

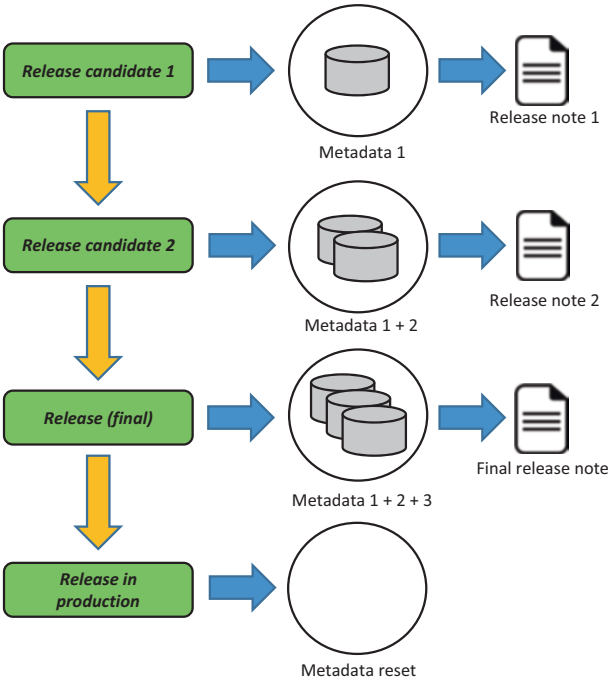


Figure 7-2. *Aggregation of release note data*

Because a release note potentially contains all features of the artifact and the associated test results, its creation should typically be done after all tests are performed. The metadata consisting of all features is generated in the *Publish artifact* stage, in which all data of the artifacts' changes and features are gathered. The *Perform test* stage generates all test results. It seems logical that the creation of the release note takes place as part of the *Notify actors* stage.

Consider this case:

A team wants to automate the creation of a release note. They use a separate issue tracker system to register the work items. Code is stored in Git, and artifacts are stored in an artifact repository.

The team is informed about each production deployment using an email (both successful and unsuccessful deployments).

Release notes are published on a wiki page. The team wants to have an aggregated release note, containing all features since the last release was deployed to production, including the test results of the last release.

A typical BPMN model could look like Figure 7-3.

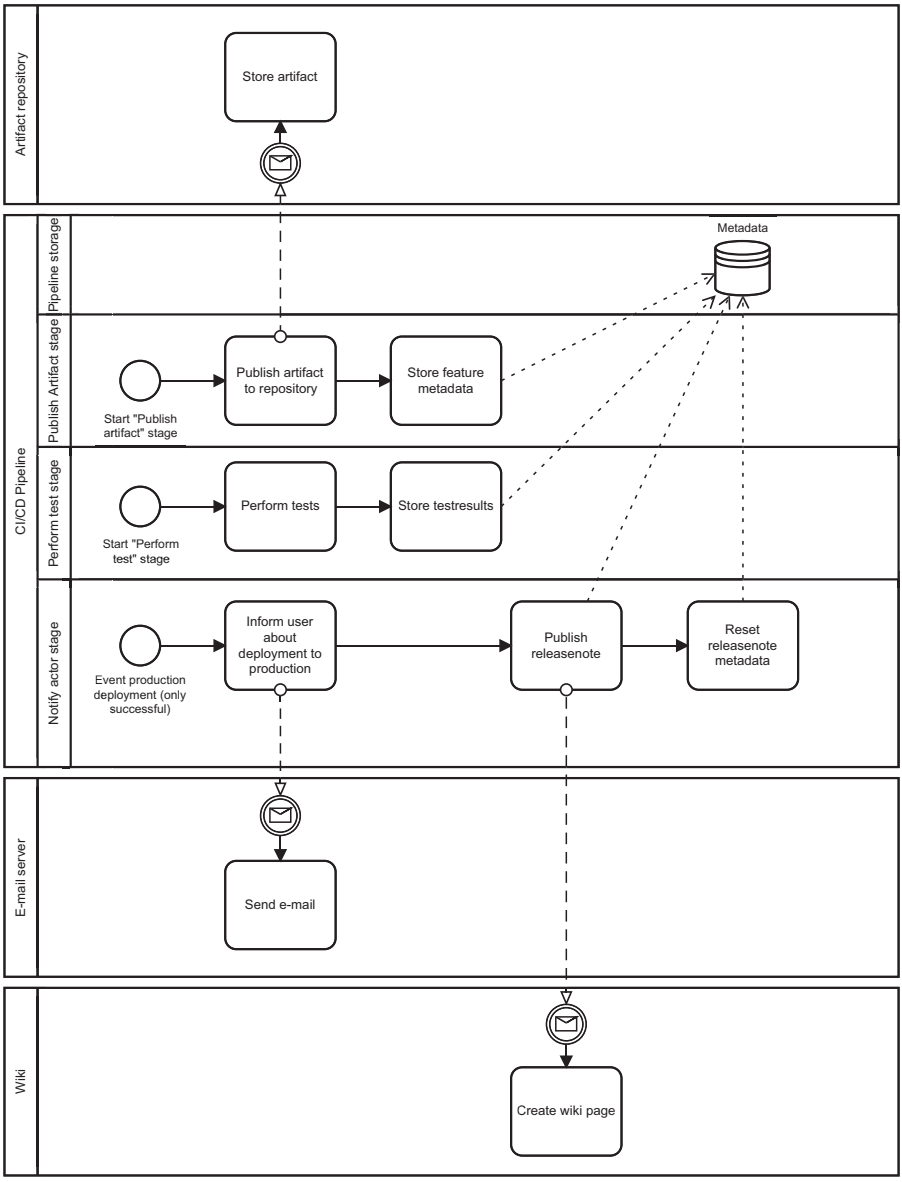


Figure 7-3. *BPMN, creating release notes*

When the *Publish artifact* stage is executed, the artifact is stored in an artifact repository, after which a specific task assembles all data associated with the artifact. This means the code commit message is retrieved and the work items associated with the release artifact are retrieved (not present in the diagram for clarity reasons). This information is stored in a database so it can be used later.

At the end of the *Perform test* stage, the test results are known. The data of the test results are gathered and also stored in the same database.

At a certain moment in the CI/CD process, the artifact is (successfully) deployed to a production environment. The result of the deployment is passed to the *Notify actors* stage, and the *Publish release note* task retrieves the metadata from the database, assembles the data, formats it to a release note, and publishes it to a wiki page. After this has been done, the metadata in the database is reset to the new start situation.

Artifact Promotion

The result of the build, package, and publish stages is an artifact stored in a binary repository. This artifact is a release candidate, meaning that it potentially can be deployed to production. But first, it has to run through various test cycles, so anything can happen along the way. During the test process, the artifact moves near production, but only a successfully tested artifact is allowed to be deployed to production. Release candidates that get stranded somewhere in the test process should be flagged because potentially there is a risk that the wrong release is deployed to production. The problem is that all release candidates, both the ones that failed the tests and the ones that passed the tests, are kept in the same binary repository. It must be possible to make a distinction between failed release candidates and successful releases. To make sure that release candidates that failed during testing are prevented from being deployed to production, a quality gate can be added; this is an additional check to determine that the artifact is valid. This check can be implemented in the *Validate exit criteria* stage.

But based on what information does this quality gate work? There are a couple of options to prevent the wrong release from being deployed.

- The artifact is promoted from stage to stage. One type of implementation is that the artifact moves through different binary repositories. So after integration testing, acceptance testing, and performance testing, the artifact is moved from one repository to the next. The last repository contains the production-ready releases, so that is the repository used in the *Deploy artifact to the production* stage. An extra condition/quality gate is not even needed because the correct repository is already used. A big disadvantage of this solution is that multiple repositories are required and the artifact is moved several times.
- Another option is to manually promote an artifact. This feature is offered by some ALM platforms. The problem with this option is that it is a manual action. A user must actively change the status of an artifact from prerelease to release, for example. The dual control stage is already a manual action, so what is the point to add more of them? To be honest, manual artifact promotion is something to avoid.
- Instead of dragging the same artifact through different repositories, there are also options to keep all artifacts in the same repository and provide metadata. After specific stages and tasks are finished and testing was successful, the metadata of the artifact is updated (using `curl` or Maven, for example). Based on its metadata, the status of the artifact is clear.

Figure 7-4 represents a unit test framework artifact with additional metadata in the format of an XML file. The metadata file (`unittest-1.0-metadata.xml`) contains additional information about the status of the test tasks.

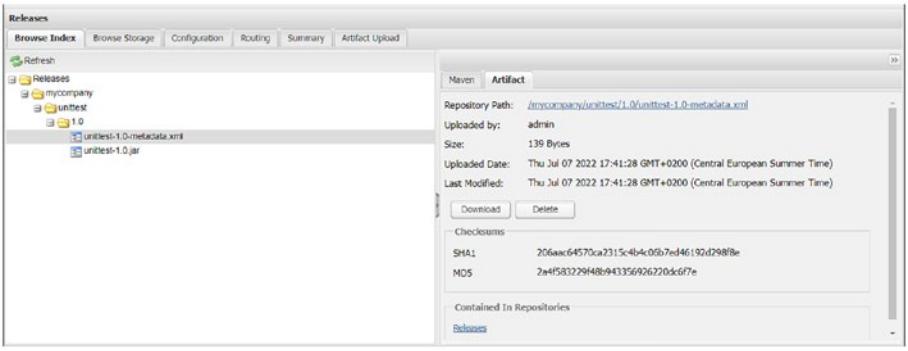
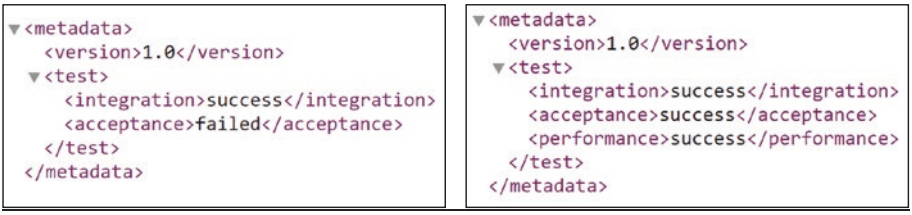


Figure 7-4. *Storing additional metadata*

Before the artifact is downloaded from the artifact repository and deployed to production, its metadata is read and interpreted (using a *quality gate* task in the *Validate exit criteria* stage). Because the acceptance test in the metadata on the left of Figure 7-5 indicates that the acceptance test failed, the pipeline ends here, and deployment to production does not take place. In the second example, on the right, the metadata shows that all tests were successful. The quality gate is passed, and the deployment can start.



unittest-1.0-metadata.xml
Release failed the tests

unittest-1.0-metadata.xml
Tests are completed successfully

Figure 7-5. *Storing metadata with test results*

Note This is less of an issue if both the CI and CD parts are implemented as one physical pipeline. The pipeline already fails before the *Validate exit criteria* stage is reached. However, it is a risk if the implementation consists of separate pipelines. One example in which this is an issue is in the case of a multiteam build strategy, in which there is one separate CD pipeline, processing artifacts from multiple CI pipelines.

Summary

You learned about the following topics in this chapter:

- Prepare the team before a pipeline is implemented and used.
  - Known issues and limitations of a pipeline should be logged and possible gaps and improvements evaluated.
  - Requirements that were not realized must be registered, including mitigating actions.

    - Address persistent problems in teams (team discipline).
- Depending on the type of integration infrastructure, a team has more or fewer responsibilities. A few operational models were covered.
    - SaaS solution
    - IaaS solution
    - Self-hosting solution
- If the pipeline is implemented, more application runtime environments need to be configured.
- Consider the use of additional features such as runbooks, release notes, and artifact promotion to improve the application implementation experience.

CHAPTER 8

Operate and Monitor

This chapter covers the following:

- How pipelines can be of use in operational processes
- The importance of monitoring pipelines and which types of monitoring are distinguished
- Some examples of the types of monitoring
- How information overloading can be reduced and how information can be presented, using different viewpoints

This chapter discusses what it takes to maintain a pipeline compared to maintaining an application.

Manage the Integration Platform

The previous chapter discussed the activities involved with operating models and setting up the integration platform, so we won't repeat those topics here. Summarized, the following operating models were identified:

- *SaaS solution*: The provider of the ALM/integration platform manages the platform, and the DevOps team can focus on developing automated pipelines.

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_8

- *IaaS solution*: The provider of the infrastructure manages the infrastructure, while the DevOps team (or IT4IT team) manages the integration platform middleware.
- *Self-hosting*: The organization is completely responsible for managing the infrastructure and the integration platform middleware.

In addition to the initial setup of the infrastructure and platform, the DevOps/IT4IT team also has to operate, maintain, and monitor the platform. It is important to emphasize that depending on the chosen operating model, this can take a significant amount of time and effort from the team.

Operational Pipelines

Pipelines are often explained in the context of building, testing, and deploying an application, but there are plenty of other areas in which pipelines also play a role. They are not necessarily CI/CD pipelines, but just pipelines used for different purposes. One area in which the use of pipelines is beneficial is in performing operational tasks associated with maintaining an application. Various activities are needed to keep the application running. These tasks should be automated as much as possible. Manual operational tasks should be discouraged for several reasons. Automating tasks speeds up operational activities and makes them repeatable, which results in more predictable results. In addition, an automated task is more secure because nobody touches the production environment with their hands. Here are some examples of operational pipelines:

- *Check for almost expired certificates*: The pipeline determines the expiration date of certificates according to a daily schedule. If a pipeline is almost expired, an alert is raised to inform the team that a certificate renewal is needed.
- *Renew certificates*: To extend on the previous bullet, a more sophisticated pipeline not only warns the team about the expired certificate but also retrieves a new certificate and installs it in the target environment. Any party interested in this renewed certificate is automatically informed, preferably using a publish/subscribe mechanism or, as an alternative, by sending an email.
- *Infrastructure drift detection*: Drift detection means that the target infrastructure has been changed compared to the infrastructure code. This is called *drift*, a topic explained in the paragraph about 'Security Monitoring'. There are multiple ways to detect infrastructure drift and to warn a team if such a thing happens. One way is to trigger a function, which detects whether infrastructure drift happened. This function is invoked using a pipeline.
- *Any other repeating function*: In addition to the previous cases, a pipeline can be used for any repeating operational function. Think about scanning a production environment for security vulnerabilities, backing up databases, performing health checks, and checking deployed artifacts in production to determine whether they are not compromised (e.g., by continuously validating the digital signature of the artifact).

- *Configure parameters*: Any parameter used by a running application must be externalized, meaning that it is not hard-coded. The application reads the parameter using a file, configuration service, or database table. To "upload" these parameters, an operational pipeline can be used.
- *Upload secrets to a vault*: Sometimes secrets like tokens or database credentials need to be uploaded from a source location to a target vault. A pipeline can be used for this.
- *Manage patches*: This means installing infrastructure patches.
- *Clean up the test environment*: The pipeline contains scripting to remove unused resources from a target environment. AWS stacks, for example, are hard to delete manually if they have dependencies with certain resources. S3 buckets with versioning enabled are typically tough to remove by hand. This can be automated and embedded into an operational pipeline.

Note The number of operational tasks is infinite. The recommendation is not to create one operational pipeline per activity but to add related tasks to one operational pipeline and make it possible to select a specific task at the start of the pipeline.

Monitor

There is not much difference between monitoring an application running in a target environment and monitoring the integration platform and its pipelines. In both cases, similar characteristics are monitored. Is the infrastructure healthy? Does the application or pipeline perform well, or is there a security issue detected? In addition, you may want to monitor certain business key performance indicators (KPIs), such as what is the success rate of the pipeline runs, or how long does it take between a work item being worked on and the actual deployment of the feature associated with this work item? Generalized, monitoring falls into a few categories.

- *Systems monitoring*: The infrastructure of the underlying ALM/integration platform is monitored. This is a type of technical monitoring that covers CPU, disk and memory usage, network congestion, etc.
- *Platform monitoring*: This can be considered an extension of systems monitoring. It covers monitoring the middleware layer of the ALM/integration platform, including the pipeline performance, health, and queuing status.
- *Business monitoring*: This covers monitoring KPIs and relates to metrics of the CI/CD process. It monitors the functional and process behavior of the platform and the pipelines. Monitoring KPIs is very specific to a team's needs.
- *Security and compliance monitoring*: This has the responsibility of monitoring all security-related aspects of the platform and pipelines. Pipeline compliance monitoring is part of this.

Several websites suggest the top four, six, or ten metrics that you should monitor. This is arbitrary and should be taken with a grain of salt. In general, you should always monitor aspects of all categories, such as the technical health of the system, the performance of the system and pipelines, and, in the case of organizations with tight security requirements, the monitoring vulnerabilities or other security-related aspects of the system. Concerning KPIs, it is up to the team what they think is important for them. So, no recommendation is given here.

Systems Monitoring

If the team or organization manages its integration infrastructure, systems monitoring must also be organized. Systems monitoring is used to validate whether the infrastructure is still healthy, but it is also used to determine whether pipelines still run in a decent and fast manner. Bottlenecks in the infrastructure have an immediate effect on pipeline execution.

Systems monitoring is arranged around various system metrics, such as the following:

- CPU usage
- Memory usage
- Disk usage
- Network usage, like HTTP sessions and HTTP response times
- Errors and logs
- Threads and processes

Any anomaly in the behavior is detected by the monitoring and alerted back to the team. The following case shows why systems monitoring is of great importance. It shows how CPU usage and the number of executors on a Jenkins server influence the performance of pipelines.

The Jenkins pipelines of a team run on a (one) Windows server, with two CPU cores. No additional nodes are used. The number of executors is set to the default; the pipelines run with two executors. The Jenkins server runs six pipelines together, implementing a payment processing system. They developed the following pipelines:

- *Receive Payment*
- *Process Payment*
- *Process Booking*
- *Book Order*
- *View Payments*
- *Inform Customer*

The pipelines represent six microservices that the system comprises. Each pipeline includes all stages to build, test, and deploy a microservice. As soon as a code commit occurs, one of the corresponding pipelines is triggered.

Baseline monitoring of the Windows server reveals that everything works fine. CPU usage is 45.12 percent, and memory usage is 34.45 percent. This is without any pipeline running. See Figure 8-1.

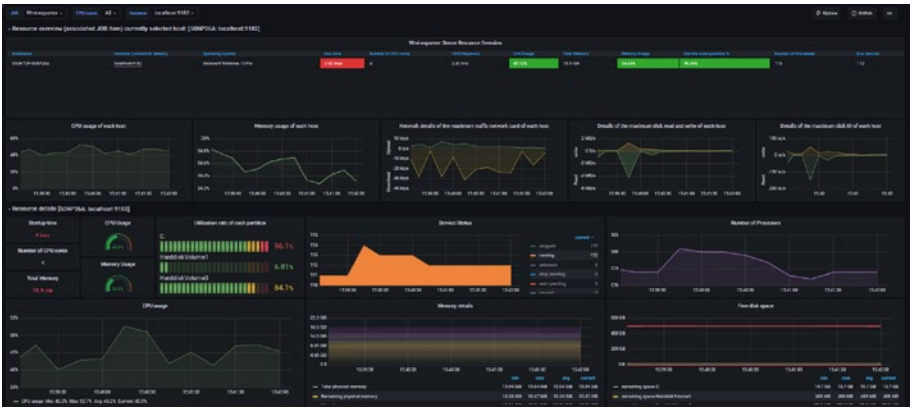


Figure 8-1. *Baseline systems monitoring*

As soon as all the pipelines are triggered at the same time, CPU usage increases but stays between 90 percent and 95 percent. Memory usage increases only slightly and stays between 36 percent and 37 percent. CPU usage is a bit on the high side, but the system is still perfectly able to run the pipelines. Table 8-1 shows the results.

Table 8-1. *Jenkins Pipeline Runs with Two Executors*

Pipeline	Start Time with 2 Executors	Execution Time with 2 Executors	Total Time Until Completed (= Start Time + Execution Time)
Book Order	After 0 sec	7 min, 31 sec	7 min, 31 sec
Inform Customer	After 0 sec	8 min, 25 sec	8 min, 25 sec
Process Booking	After 7 min, 31 sec	8 min, 57 sec	16 min, 28 sec
Process Payment	After 8 min, 25 sec	5 min, 5 sec	13 min, 30 sec
Receive Payment	After 13 min, 30 sec	4 min, 40 sec	18 min, 10 sec
View Payments	After 16 min, 28 sec	9 min, 26 sec	25 min, 54 sec

What stands out is that if all pipelines are triggered at the same time, not all pipelines start immediately, and the last pipeline (View Payments) is finished only after 25 minutes and 54 seconds. This means the developer receives information about the pipeline execution after more than 25 minutes since the code was committed and pushed. This is not a surprise because the number of executors is set to two, meaning that only two pipelines are executed at the same time. The other pipelines become pending until one of the executors is available again. This is problematic if the commit rate is high because each commit triggers a pipeline.

No problem, you would say. Just increase the number of executors to, let's say, four. This changes the results slightly, as shown in Table 8-2.

Table 8-2. *Jenkins Pipeline Runs with Four Executors*

Pipeline	Start Time with 4 Executors	Execution Time with 4 Executors	Total Time until Completed (= Start Time + Execution Time)
Book Order	After 0 sec	9 min, 19 sec	9 min, 19 sec
Inform Customer	After 0 sec	10 min	10 min
Process Booking	After 0 sec	14 min	14 min
Process Payment	After 0 sec	8 min, 10 sec	8 min, 10 sec
Receive Payment	After 8 min, 10 sec	6 min, 12 sec	14 min, 22 sec
View Payments	After 9 min, 19 sec	9 min, 2 sec	18 min, 21 sec

The start time of the last pipeline (View Payments) is reduced, from 16 minutes and 28 seconds to 9 minutes and 19 seconds. That is an improvement, but the overall execution time of most individual pipelines is

increased[1] by a couple of minutes. The total time of executing all pipelines, however, reduces from 25 minutes 54 seconds to 18 minutes 21 seconds.

The CPU capacity is spread over multiple pipelines. This is visible in the systems monitor as showed in Figure 8-2, which stays most of the time at 100 percent. Memory usage is still low between 36 percent and 37 percent, meaning that the pipelines are CPU bound and not memory bound. See Figure 8-2.

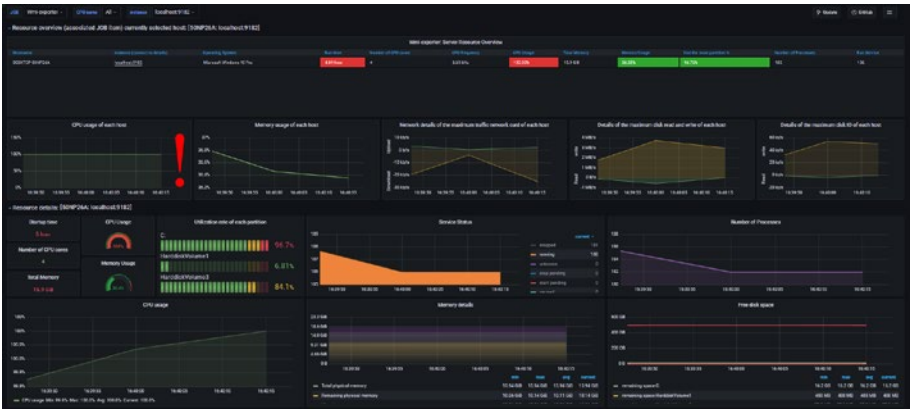


Figure 8-2. *Systems monitoring with 100 percent CPU usage*

Increasing the number of executors smooths out the *Total time until completed*, but it comes with a price. The overall execution time of the pipelines in concurrent runs increases. This is even more dramatic if the number of executors is increased to six and all pipelines are started at the same time. See Table 8-3.

[1] Note that the View Payments is even faster with four executors instead of two. This can be explained because when the pipeline starts, most other pipelines are already finished, so this pipeline has more CPU resources at its disposal.

Table 8-3. *Jenkins Pipeline Runs with Six Executors*

Pipeline	Start Time with 6 Executors	Execution Time with 6 Executors	Total Time until Completed (= Start Time + Execution Time)
Book Order	After 0 sec	12 min	12 min
Inform Customer	After 0 sec	13 min	13 min
Process Booking	After 0 sec	16 min	16 min
Process Payment	After 0 sec	11 min	11 min
Receive Payment	After 0 sec	10 min	10 min
View Payments	After 0 sec	16 min	16 min

Playing with the number of executors results in a shift regarding *Execution time* and *Total time until completed*. If the number of executors is low, CPU utilization is optimal, resulting in a faster execution time. But if the number of commits becomes higher, you need to make a choice. With fewer executors, *Execution time* of the running pipeline instances is optimal, but other pipeline instances start to queue. You might want to increase the number of executors to spread the CPU resources evenly over the running pipeline. This reduces the *Total time until completed* but does increase the *Execution time* value of all pipelines.

But what if you want a lower *Execution time* but also a lower *Total time until completed*? The only option is to add more computing capacity because the systems monitor indicates that CPU usage is a bottleneck. After all, it continuously stays at 100 percent. Adding more capacity can be achieved by adding more nodes (servers) and offloading pipeline runs to these nodes so the main Windows server capacity is freed up.

This case shows how to play with the number of executors, and it is a nice example of using systems monitoring to spot bottlenecks in pipeline processing.

Platform Monitoring

Platform monitoring is positioned one level above infrastructural systems monitoring. Platform monitoring concerns the monitoring of the ALM/integration platform itself. This includes the platform middleware and the pipelines. The following are the typical metrics to monitor:

- Queue depth of all nodes/servers/agents (to detect queuing/pending pipelines).
- Performance of pipelines.
- Number of pipeline runs.
- Number of successful and failed pipelines related to infrastructure problems or issues with external connections.
- Team behavior; a team scheduling thousands of jobs in a very short time creates a bottleneck for teams that continuously—but with a low pace—start their pipelines. When monitoring this, it becomes possible to address the teams about it (or possibly apply a certain concurrency policy, if that feature even exists…!).

To be honest, most ALM/integration platforms provide poor support for dashboards that monitor platform-specific metrics. In general, a lot of improvements can be made in this department.

Take a look at Figure 8-3. It shows a simple build and deployment health dashboard, including statistics on the number of (partial) successful, failed, and canceled pipeline runs in the last 90 days. In addition, the status of the latest pipeline runs is visible.

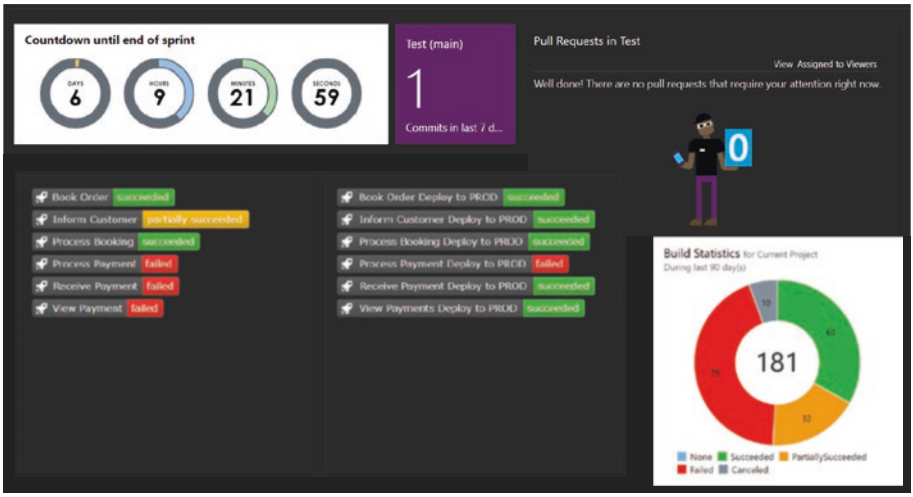


Figure 8-3. *Simple health dashboard*

Although this dashboard gives some insight into the latest pipeline runs, it is still a rudimentary dashboard, and in this particular case, it was not possible to configure a dashboard in such a way that it fulfilled all the requirements, especially information about pipeline performance is omitted. Metrics like *what is the average processing time?* of the various pipelines and *how does it change over time?* are hard to monitor. Also things like *how long does a pipeline run remain in the queue before it is executed?* and *what are the maximum and average queuing times?* are problematic to monitor, or at least difficult to display in a dashboard.

In general, the requirement to spot any degradation or bottleneck in pipeline processing because of infrastructure/platform issues was difficult to be fulfilled with the standard options available for the various analyzed platforms.

Business Monitoring

KPIs can be visualized using custom dashboards. A few examples of KPIs were mentioned in Chapter 3. The next dashboard example visualizes two KPIs called *Lead time* and *Cycle time*. These KPIs need some explanation.

- *Lead time* is the time measured from the moment a work item is created and the moment it becomes in a final state (Done). During this time, the code associated with this work item is developed and tested. The work item status is set to *Done* after all tests have been completed.

  Lead time does not say anything about the performance of the DevOps team. The time between the moment a work item is created and the moment it is pulled into a sprint and picked up by a developer can be very long. A work item can stay on the backlog for a very long time.

- *Cycle time* gives better insight into the performance of the team. It measures the time between a developer committing themselves to a work item and the moment the code for the particular feature has been developed and tested.

Figure 8-4 visualizes the difference between *Lead time* and *Cycle time*.

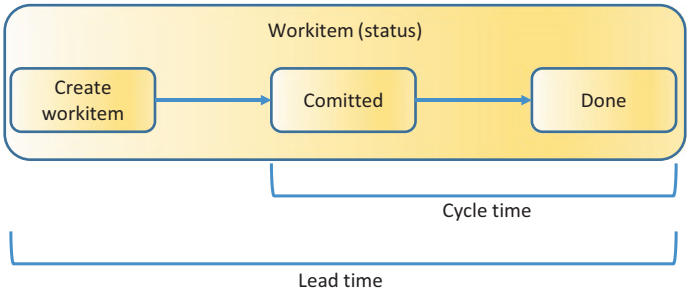


Figure 8-4. *Lead time and Cycle time*

The dashboard shows both KPIs. The average *Lead time* is 91 days, based on 18 work items (the 10 bugs excluded), while the *Cycle time* is 23 days on average. This means the 18 work items stayed on the backlog for 68 on average, while the team finished a feature in 23 days on average.

Are there any conclusions to be made, based on this dashboard? The fact that a work item stays on the backlog for more than two months does not say anything. Probably there was no real urgency to solve these items. But based on a *Cycle time* of 23 days, we can conclude a couple of things because it takes a relatively long time to finish these work items.

When zooming in on the dashboard, a couple of work items really stand out. The red-circled dots on the dashboard are work items with a *Cycle time* of, respectively, 125.2 days, 74.8 days, and 63.9 days. These work items influence the average *Cycle time* negatively. Detailed inspection reveals that these work items include activities that are performed by another department but are required to finish the work item. It is not the DevOps team to blame for the delay, and it may give the wrong impression of the team's performance; a careful analysis is required before any conclusion is made.

However, a few conclusions can be made. Splitting work items into activities performed by the team and activities performed by another department would have contributed to a more accurate representation of the teams' velocity. But even if the outliers are removed, the average *Cycle time* is still a couple of days on average. These numbers can be discussed with the team. See Figure 8-5.

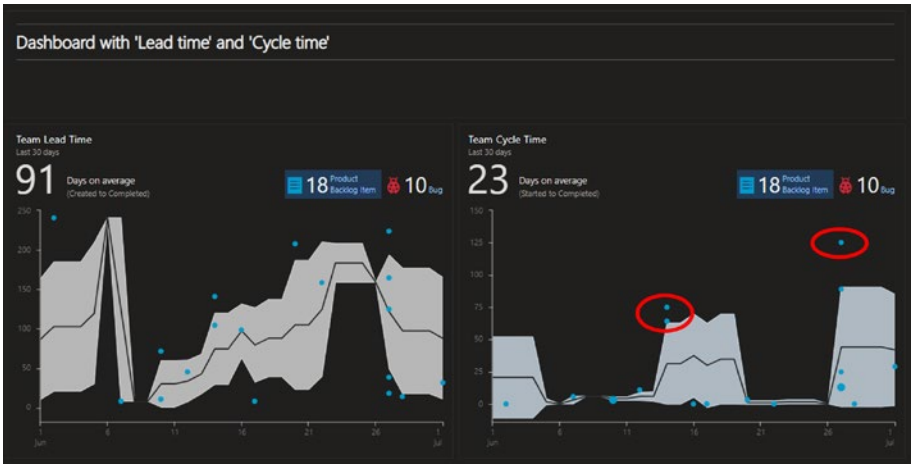


Figure 8-5. *Dashboard, Lead time and Cycle time*

Security Monitoring

Security monitoring covers a broad range of topics. The integration platform and infrastructure must be secure, and any vulnerabilities or breaches must be detected by the monitoring systems. In addition, various checks can be done on the pipelines themselves. For example, a pipeline has to comply with the company policies. So, let's zoom in on two examples.

Application monitoring and monitoring of the target environment on which the application runs are typically not part of integration platform and infrastructure monitoring. However, there are a few types of monitoring that do fall into this category. Consider an application deployed to a certain target environment. The application may not be altered once deployed, and if it is changed, it can be changed only using a pipeline redeploy and not manually. The same applies to the target environment itself. Once the infrastructure has been provisioned and applications run on it, any manual change of the infrastructure is not allowed and should be detected. This type of monitoring can be considered part of pipeline (security) monitoring.

Figure 8-6 shows an example in which part of the infrastructure—a stack—is provisioned to an AWS account. The infrastructure is provisioned using IaC, and once provisioned, it can be changed only by re-provisioning the updated infrastructure code. In this particular screenshot, the stack is changed manually, indicated by the *Drift status*. It has the value `DRIFTED`, while the default *Drift status* should be `IN_SYNC`.

Continuous monitoring of the infrastructure drift or changes in the applications deployed on this infrastructure is a good way to detect any manual change in the production environment. A cloud service provider like AWS has the tools to check for drift of both infrastructure and

applications,[2] and continuous monitoring can be done relatively easily. Any infrastructure drift or nonauthorized application changes are exposed on the AWS console, as depicted in Figure 8-6.

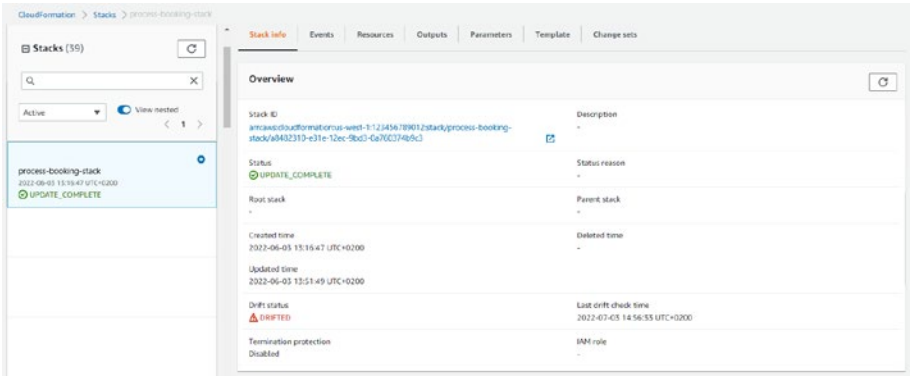


Figure 8-6. *Drift detection of an AWS stack*

Detecting drift of the AWS infrastructure is an automated process, but it is not triggered automatically. Encapsulating the drift detection function in a scheduled operational pipeline is one way to detect and monitor drift.

Compliance monitoring validates whether the platform and pipelines meet the company policies. Any deviation results in a noncompliant flag. This could mean that the team is just informed about the fact that certain parts of their setup or pipelines are not compliant, but the compliance checks could also have a mandatory character. If the pipeline is not compliant, it is blocked from execution.

The Pipeline Compliance Dashboard in Figure 8-7 shows various policies that indicate whether a pipeline is compliant. A regular scan is performed to update the dashboard with the latest information. DevOps

[2] Lambda code signing is a way to determine whether running code has been altered.

teams can view the compliance status of their pipelines. In this particular example, the pipeline is not compliant because the infrastructure validation task is omitted from the pipeline. A short explanation of the problem and the solution are given, as shown here:

> *This pipeline does not have an 'AWS Infrastructure scanning' stage*
>
> *A production environment must be configured in such a way that it meets the company security policies. Add the IT4IT AWS Infrastructure scanning task 2.0 to your pipeline to scan your infrastructure code and test compliance of the pipeline using the Validate button.*

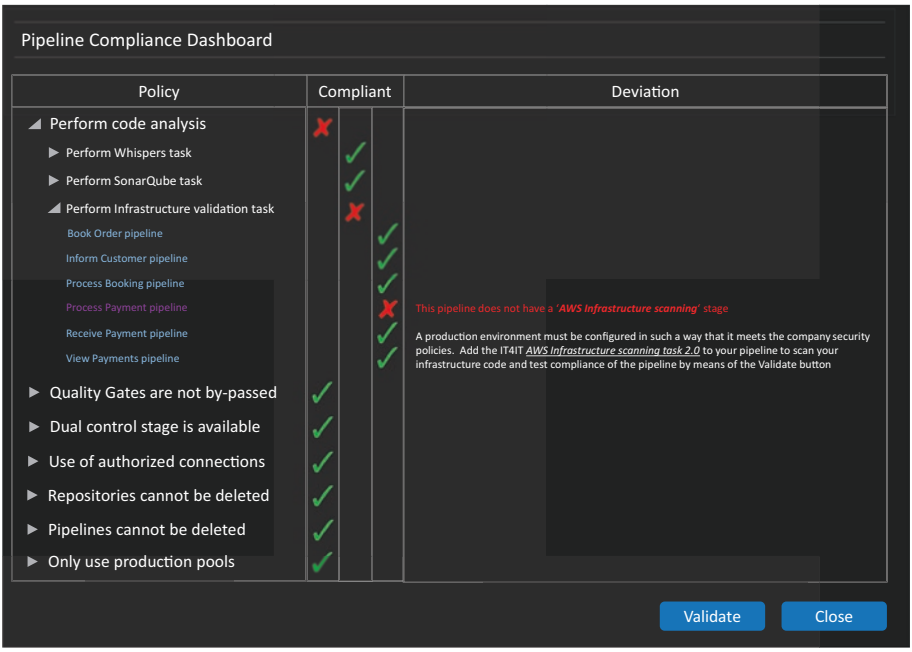

Pipeline Compliance Dashboard
Policy
Compliant
Deviation
Perform code analysis
Perform Whispers task
Perform SonarQube task
Perform Infrastructure validation task
Book Order pipeline
Inform Customer pipeline
Process Booking pipeline
Process Payment pipeline
Receive Payment pipeline
View Payments pipeline
Quality Gates are not by-passed
Dual control stage is available
Use of authorized connections
Repositories cannot be deleted
Pipelines cannot be deleted
Only use production pools
This pipeline does not have a 'AWS Infrastructure scanning' stage
A production environment must be configured in such a way that it meets the company security policies. Add the IT4IT AWS Infrastructure scanning task 2.0 to your pipeline to scan your infrastructure code and test compliance of the pipeline by means of the Validate button
Validate
Close

Figure 8-7. *Pipeline Compliance Dashboard*

Share Information

Information can be shared in different ways, but beware that information overloading of the DevOps team must be prevented. The best way to demonstrate what an "information sharing" design could look like if techniques to prevent information overloading are applied is by using a specific case. Of course, this case depicts only one possible solution, and teams have to decide for themselves what their information flow will look like. Consider the following case:

- *A team uses a feature branch workflow. It makes use of Microsoft Teams and email to inform the team.*
- *In the case of a feature branch, the results of the build and unit test stages are sent using an email to the concerned developer only.*
- *In the case of the trunk (main branch), the pipeline creates a release. The following requirements apply:*
  - *The result of a release build, both successful and unsuccessful, must be sent to a specific channel in Microsoft Teams called release build.*
  - *The result of all tests (including unit tests), both successful and unsuccessful, must be sent to a specific channel in Microsoft Teams called test.*
  - *The result of a production deployment, both successful and unsuccessful, must be sent to a specific channel in Microsoft Teams called production deployment.*

- *If a dual control must be performed, an email is sent to the product owner only; a delegate can view the product owners' mailbox.*
- *If a production deployment fails, all team members are informed about the result using an email. They will not get any email if the production deployment is successful.*

Given these requirements, a design is drafted. The team's branching strategy is defined as a feature branch workflow. A typical BPMN model looks like Figure 8-8.

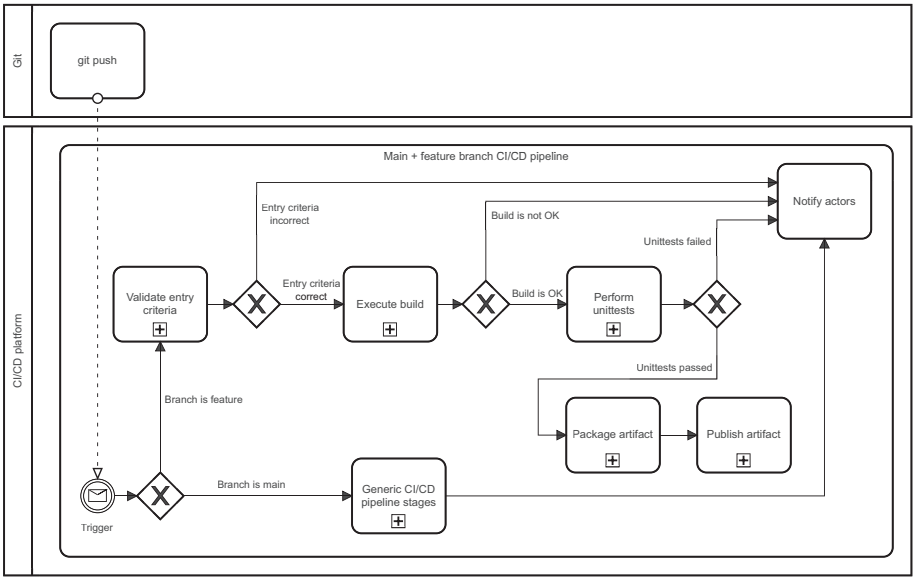


Figure 8-8. *BPMN, sharing information*

It shows the feature branch workflow with a feature branch and a trunk (the *main* branch). The *Notify actors* stage is responsible for communication with other actors, and the requirements state that both successful and unsuccessful results must be communicated. This explains the presence of the parallel gateway after certain stages. Also note that the diagram does not have an end event; start and end events in BPMN are optional.

The results of executing a stage are passed as arguments to the *Notify actors* stage. *The following are the i*nput arguments of the Notify actors:

- The developer who performed the code push.
- The repository branch.
- The executed stage is passed as an argument. This stage is called previous_stage in Figure 8-9.
- The results of the stage, success or failure.

If we zoom in on the *Notify actors* stage, it results in a detailed design, as depicted in Figure 8-9.

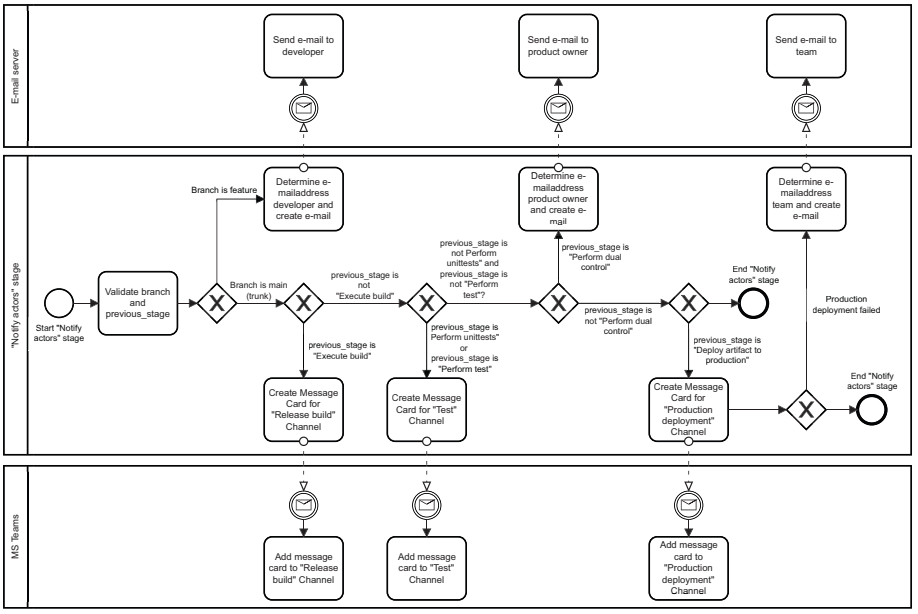

E-mail server
Send e-mail to developer
Send e-mail to product owner
Send e-mail to team
"Notify actors" stage
Start "Notify actors" stage
Validate branch and previous_stage
Branch is feature
Branch is main (trunk)
Determine e-mailaddress developer and create e-mail
previous_stage is not "Execute build"
previous_stage is "Execute build"
previous_stage is not Perform unittests" and previous_stage is not "Perform test"?
previous_stage is Perform unittests" or previous_stage is "Perform test"
Determine e-mailaddress product owner and create e-mail
previous_stage is "Perform dual control"
previous_stage is not "Perform dual control"
End "Notify actors" stage
Determine e-mailaddress team and create e-mail
Production deployment failed
previous_stage is "Deploy artifact to production"
End "Notify actors" stage
Create Message Card for "Release build" Channel
Create Message Card for "Test" Channel
Create Message Card for "Production deployment" Channel
MS Teams
Add message card to "Release build" Channel
Add message card to "Test" Channel
Add message card to "Production deployment" Channel

Figure 8-9. *BPMN, sharing information (Notify actors)*

In the *Notify actors* stage, the first validation is on the branch. A different path is followed for the main branch compared to the feature branches. In the case of a feature branch, an email is created for the developer and sent to the developer using an email server. The path of the feature branch stops here.

In the case of the main branch, multiple validations are performed. The previous_stage (passed as an argument to the *Notify actors* stage) is validated, and based on its value, either emails or message cards in Microsoft Teams are created.

Events, Alerts, Incidents, and Notifications

The following are the events, alerts, incidents, and notifications you'll see:

- An event is an occurrence of a situation in the system that takes place. It can be a certain metric exceeding a threshold, but it can also be a state change in the system. If a pipeline fails, it results in the submission of an event; if storage usage exceeds 80 percent, it results in an event, an unhealthy pipeline results in an event, etc.
- A notification is a message to inform the user about a certain—noncritical—event that occurred. The creation of a release note is not critical, but perhaps it's important enough to share with the team.
- An alert is an urgent notification, triggered if a certain event (or multiple events) takes place with a certain importance. Storage usage exceeding 80 percent is important enough to be shared with specific people from the team.
- An incident is an alert that causes damage or is a threat to the system. It is of utmost importance to push this information to the team because it concerns a blocking issue, which causes either a serious degradation of the pipeline performance or the pipelines do not work at all.

Notifications, alerts, and incidents are shared with the team. In the case of incidents, the team should be informed proactively, based on a push mechanism; one or more team members are informed using an email, an SMS message, or a WhatsApp message because immediate action is required. Notifications and alerts can be shared by these same channels, but it is also possible to inform the team with a notification or alert on a dashboard or overview. The team members have to actively watch the dashboard to be kept informed.

In all cases, you need to be conservative with the amount of information you push to the team. Only if needed, information is actively pushed.

The overview in Figure 8-10 gives a nice example of the pipeline runs of a process booking pipeline in Jenkins. There are some issues with the latest runs. In one of these runs, the build failed. In the latest run, all stages were executed properly again; however, the *Deploy artifact to production* stage ended with a warning, although the deployment was successful. Further investigation is needed. Since the overview already gives a nice indication that something went wrong, the team has to decide whether they also want to be alerted actively or whether keeping an eye on the stage view screen is sufficient.

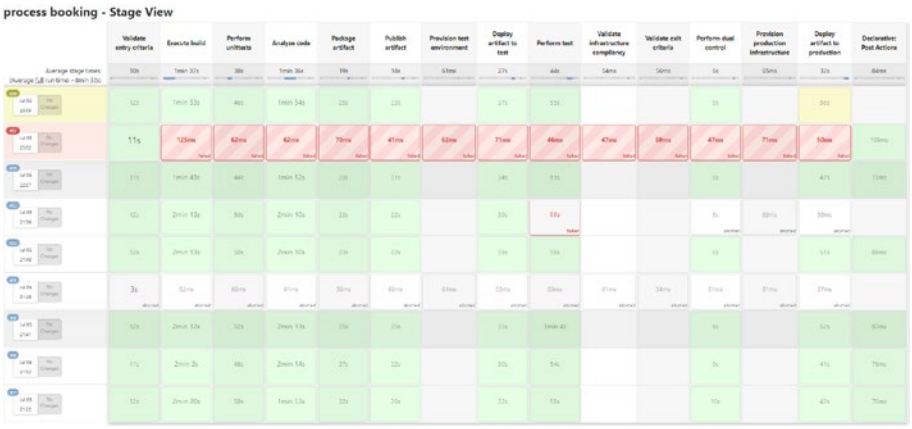


Figure 8-10. *Jenkins stage view process booking*

One problem with these dashboards is that information can be retrieved, but it takes a couple of clicks. Figure 8-11 represents the Jenkins Blue Ocean dashboard and shows the latest run of the process booking pipeline. It shows that artifact version 11.4 was deployed with a warning, but it requires navigation to this particular screen and selecting the stage for which the detailed information needs to be displayed.

Figure 8-11. *Jenkins blue ocean process booking*

But sometimes you just want to have a different view of the information. Instead of relying solely on the ALM/integration platform to provide the information, it is also possible to make use of other channels. Email, SMS, and WhatsApp were already mentioned, and they form an

excellent way to push information, but communication and collaboration platforms like Microsoft Teams are also a good addition to the way information is presented. Take a look at the overview in Figures 8-12 and 8-13. The results of the *Execute build* and *Deploy artifact to production* stages are sent to Microsoft Teams as a message card. This information is grouped into different communication channels. The *Execute build* channel contains all release build notifications, and the *Deploy artifact to production* channel contains the notifications related to production deployments. As you can also see, there are two build notifications in the *Execute build* channel. The first build failed, and the next build was successful. Build artifact version (1.1.4) was successfully deployed to production, although with warnings, which are displayed in the *Deploy artifact to production* channel. Using these kinds of tools makes it possible to arrange information differently and make it more attractive and accessible.

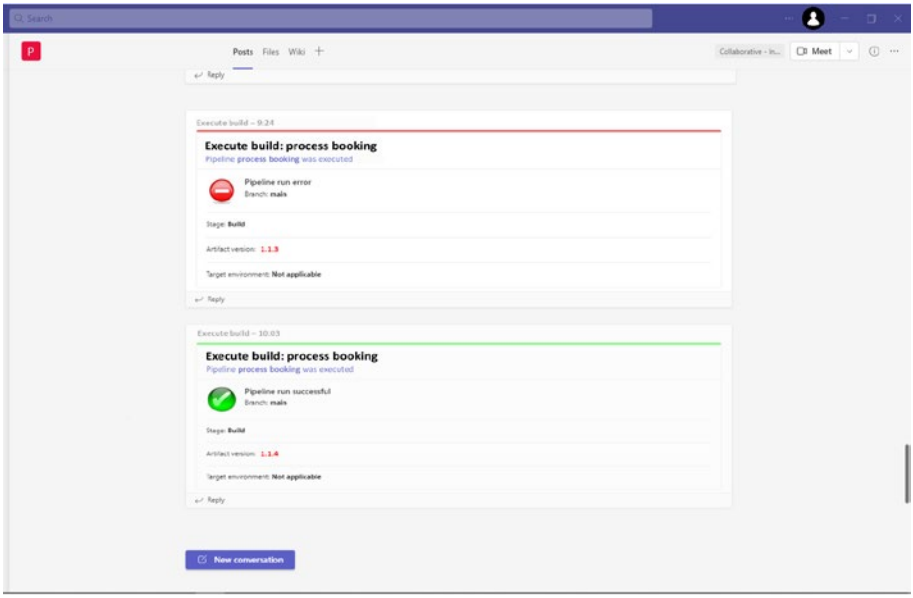


Figure 8-12. *Notifications, displayed in the Execute build channel of Microsoft Teams*

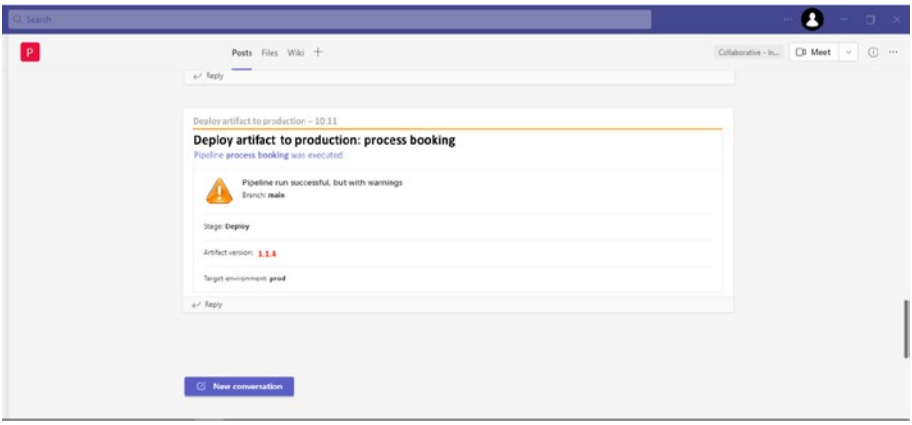


Figure 8-13. *Notification displayed in the Deploy artifact to production channel of Microsoft Teams*

Summary

You learned about the following in this chapter:

- Operating and maintaining an ALM/integration platform can put a burden on the team, depending on the operating model.
- Pipelines are very useful for regular operational tasks and recurring tasks.
- Monitoring pipelines is done on multiple levels.
  - Systems monitoring
  - Platform monitoring
  - Business monitoring
  - Security monitoring

Make sure to cover them all.

- Information overloading must be prevented. Use different communication channels to present information from different viewpoints, to make it more attractive and accessible.

CHAPTER 9

Use Case

This chapter covers the following:

- The use case of MyCorp.com and the AWSome team.
- An overview of requirements implemented by the AWSome team.
- The design of the pipeline based on the given requirements.
- An implementation of the use case using Azure DevOps pipelines.
- A detailed configuration. With this configuration, it becomes possible to execute the pipeline code, provided separately from this book.
- The result of the pipeline runs.
- The results, the gaps and backlog items, and the output of a running application.

All the source code is available in the GitHub repository: `https://github.com/Apress/Continuous-Integration-(CI)-and-Continuous-Delivery-(CD)`.

Up to now, the approach to designing and developing pipelines has been discussed on an abstract level, but it has not yet led to a real pipeline that runs on a machine. This chapter presents a use case and guides you

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0_9

through all the steps explained in this book, from requirements analysis to the implementation of a pipeline that runs on an ALM platform. Azure DevOps is the ALM platform that is used to demonstrate the use case. But even if you don't know anything about Azure DevOps or AWS, this chapter is still valuable because it shows a real case from requirements to implementation.

Note This chapter is not a tutorial on creating a pipeline in Azure DevOps, but it does guide you through the steps needed to set up the pipelines. Details of certain steps have been omitted for clarity and are believed to be familiar to readers who already have some experience with Azure DevOps. Also, the combination of Azure DevOps and AWS is not the most obvious choice, because AWS also provides the tools, but it demonstrates that you can easily use different ALM platforms.

The case deals with an imaginary company called myCorp.com. It is a new startup with several small development teams. One of these teams is the AWSome team, consisting of a product owner named Emma and three engineers, named Meera, Tim, and Vinod. Vinod is also a delegated product owner and approves or declines deployments on behalf of Emma.

The team's ambitions are huge, but they decide to start small. Their first application is called *myapp*, and the first increment consists of only a healthcheck app. It just listens to an HTTP request and logs a message if the request is processed. It's not very exciting, but the team wants to establish a solid workflow and develop their first automated pipeline.

To make a difference in the world, myCorp.com attaches great importance to sustainability. The employees do not want to set up an on-premises data center; everything is done in the cloud, and they decide to use AWS as the runtime environment for all their apps.

The journey of the AWSome team begins with a requirements analysis.

Requirements Analysis

The requirements of myapp and its first increment—the healthcheck app—are clear. The healthcheck is realized as an AWS Lambda function that listens to HTTP requests and writes a log line to a CloudWatch log after every processed request. The healthcheck Lambda is called every 5 minutes by a CloudWatch schedule.

Because the runtime environment is AWS, the team chooses infrastructure as code (IaC), but they use the AWS Cloud Development Kit (CDK) over AWS CloudFormation. By using CDK, the infrastructure is fully coded in their favorite programming language, Java.[1]

Defining continuous integration, continuous delivery, and pipeline requirements take a bit more work, so the AWSome team decides to draft a table with all the requirements; see Table 9-1.

Table 9-1. *Requirements*

Sustainability	
Define sustainability goals.	After validating several ALM platforms, the team chooses Azure DevOps. This is a cloud solution, developed by Microsoft, running on Azure.
Way of Working	
Use a simple branching strategy.	The team has experience with a feature-based branching workflow. The main line is kept in a production-ready state.

(*continued*)

[1] AWS CDK supports multiple languages.

Table 9-1. (*continued*)

Choose the release strategy you want.	The team works in sprints of two weeks and wants to deploy to production at the end of every sprint, using a timeboxed release strategy. Although they want to deploy in a fully automated way after every sprint, the decision is made to manually trigger the deployment for now.
Choose a build, test, and deployment strategy.	• The build strategy of the pipeline is kept simple. The choice is made to perform a full Maven build in each pipeline run. • Also, the deployment strategy is simple. The AWSome team starts with a re-create deployment strategy, looking into canary releases in one of the next increments. • No specific requirements for the test strategy are defined yet. The technical test framework used for application acceptance tests is Cucumber.
Technology	
Automate the creation of ephemeral test environments.	The test environment is created or adjusted in each pipeline run, using AWS CDK. For now, the test environment is not deleted after every test run.
Decide upon the development strategy.	The approach is to use the extended pipeline development strategy. However, the team does not have a unit testing framework, so they opt for a process where pipeline development and testing are done exclusively in a separate Azure DevOps test project.

(*continued*)

Table 9-1. (*continued*)

Compliance and Auditability	
All changes are traceable/ tag everything.	Tagging—using the release version—is done for each release. The following resources are tagged: • Code in Git • Pipeline(s) • Build artifacts • AWS stacks
Only build and deploy artifacts using a pipeline.	To provide evidence of the integrity of the artifact, an SHA256 hash of the artifact is generated after the build and deploy steps and compared with the hash of the lambda in the AWS account. All hashes must be the same.
Resources associated with a release cannot be deleted.	Update the retention time of resources associated with a production release (to *forever*).
Security (General)	
Refine access by setting permissions for a user or group.	Create a separate group for dual control and assign Emma and Vinod to this group.
Perform a vulnerability analysis.	The following validations are done: • Whispers for hard-coded secrets • Lambdaguard for AWS Lambda configuration

(*continued*)

Table 9-1. (*continued*)

Manageability	
Use a release versioning schema that makes sense.	Use semantic versioning for releases. Each release version must be generated.
Pipeline code is treated as software.	Pipeline code is stored in an Azure DevOps Git repository.
Store binaries in an artifact repository.	Azure DevOps has the option to store the artifact together with the build pipeline. This option is used; the team does not make use of an external repository or Azure DevOps Artifacts.
Build once, run anywhere.	The artifact is built only once and deployed to separate AWS test and production accounts.
Quality Assurance	
Application code must be scanned on code quality.	SonarCloud is used for scanning code quality.
Use quality gates.	In addition to a quality gate after the SonarCloud scan, use a quality gate just before the artifact is deployed to production to guarantee that certain stages are executed (*Analyze code*, *Perform test*, *Validate infrastructure compliance*).
Define entry and exit criteria.	Each pipeline starts with validations to determine whether variables are configured properly.

Pipeline Design

To get a clear understanding of the environments, the tools, and how everything is connected, the context diagrams in Figure 9-1 and Figure 9-2 are drafted.

The first diagram represents the Azure DevOps environment, used to run the pipelines. It consists of two projects. The application is developed in the main (production) project. This project is also used to run the pipelines and deploy them to the AWS test and production environments.

A second project—the test project—is a clone of the main project and is solely used to develop and test the pipelines.

The main Azure DevOps project is connected to both the AWS test and the AWS production environments. The Azure DevOps test project is connected only to the AWS test environment, so it cannot deploy to the AWS production environment. The AWS test environment is represented by account[2] 497562947267. The AWS production account has the ID 486439332092.

Both Azure DevOps projects are connected to SonarCloud. External libraries are retrieved from the central Maven repository, and emails are sent from the Azure DevOps pipelines to the team members.

[2] Both account numbers 497562947267 and 486439332092 are the account of the AWSome team. If you want to try the pipelines yourself, you need to request and use your own AWS accounts, of course.

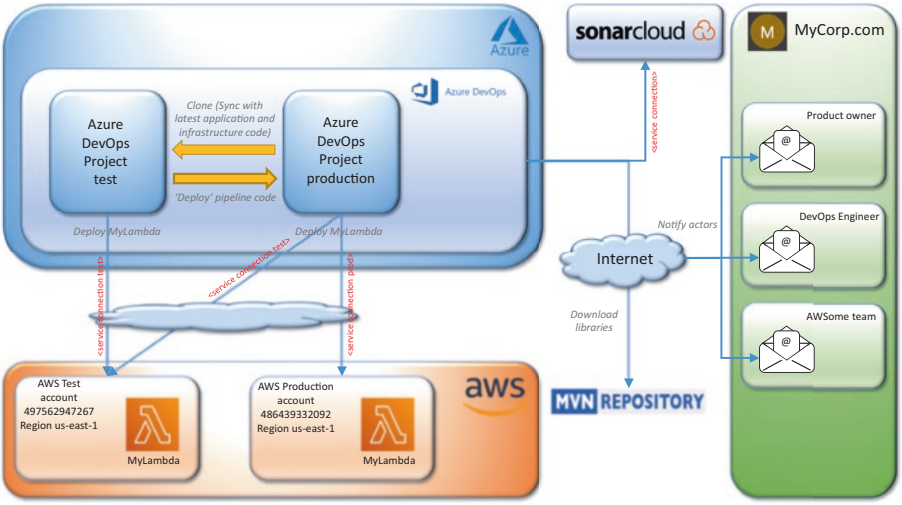


Figure 9-1. *Context diagram*

The second context diagram represents an Azure DevOps project. Each Azure DevOps project consists of Git repositories, environment configurations, service connections, permissions, and variable groups. These need to be configured before the pipeline can work.

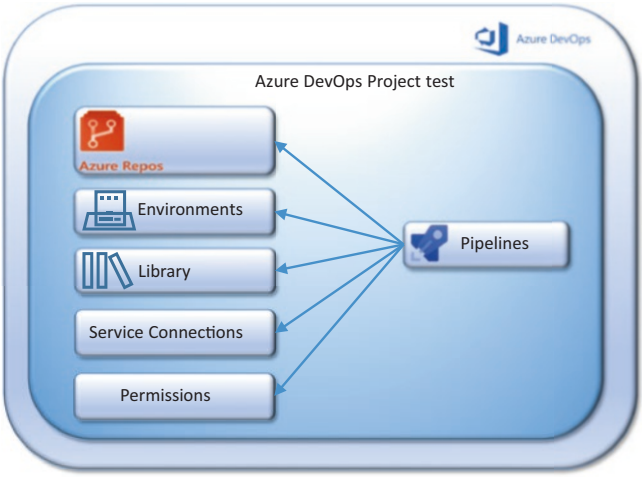


Figure 9-2. *Context diagram, Azure DevOps project*

Branching and Release Strategy

As mentioned in the requirements analysis from Table 9-1, the team decided to adopt a feature-based branching workflow in combination with a timeboxed release strategy. The branching strategy results in two pipelines, one associated with a feature branch and another pipeline associated with the main branch, as depicted in Figure 9-3 and Figure 9-4.

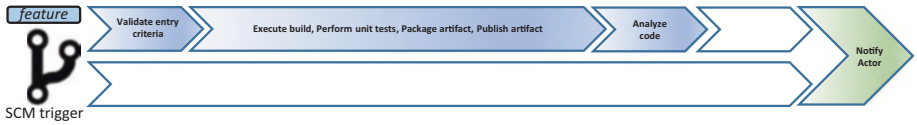


Figure 9-3. *Workflow, pipeline of the feature branch*

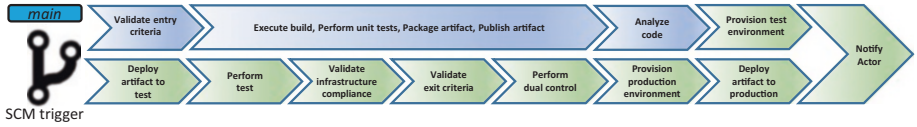


Figure 9-4. *Workflow, pipeline of the main branch*

Instead of separating the continuous integration process into individual stages, the choice is made to combine the *Execute build*, *Perform unit tests*, *Package artifact*, and *Publish artifact* stages into one stage. The reason is that the team uses Maven, which makes it easy to combine these stages in one command.

As a result of the timeboxed release strategy, the pipeline associated with the main branch is split into two separate pipelines. The first pipeline consists of the stages of the Generic CI/CD Pipeline, except for the stages associated with the deployment to production. These stages are moved to a separate pipeline. This means that based on the branching and release strategy, three pipelines are distinguished.

- Pipeline 1 is associated with the feature branch.

- Pipeline 2, the primary pipeline, is associated with the main branch. The pipeline contains the stages of the Generic CI/CD Pipeline, until the *Validate exit criteria* stage.
- Pipeline 3, the production deployment pipeline, is associated with the main branch. The pipeline contains the stages of the Generic CI/CD Pipeline, starting from the *Validate exit criteria* stage.

The latter two pipelines are represented in Figure 9-5 and Figure 9-6.

Figure 9-5. *Workflow, primary pipeline*

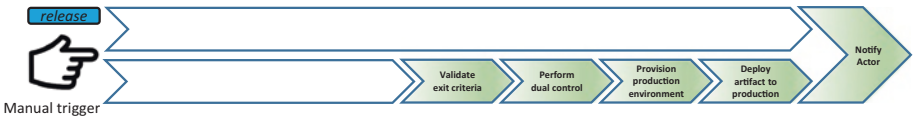


Figure 9-6. *Workflow, production deployment pipeline*

The three pipelines are logical pipelines and will be implemented by technical pipelines. Pipelines 1 and 2 are combined in one pipeline called *myapp-pipeline*; they cover all continuous integration activities and all activities associated with testing. For performance reasons, the team decides to execute the *Analyze code* stage in parallel with the *Execute build* stage. The other pipeline—*myapp-production-deployment*—covers all activities dealing with the deployment to production. Both pipelines are represented in the BPMN models shown in Figure 9-7 and Figure 9-8.

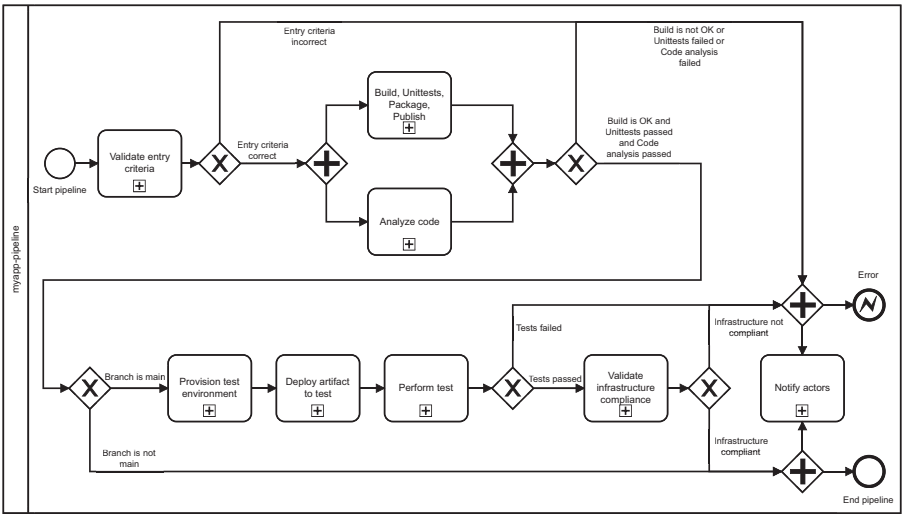


Figure 9-7. *BPMN, myapp-pipeline*

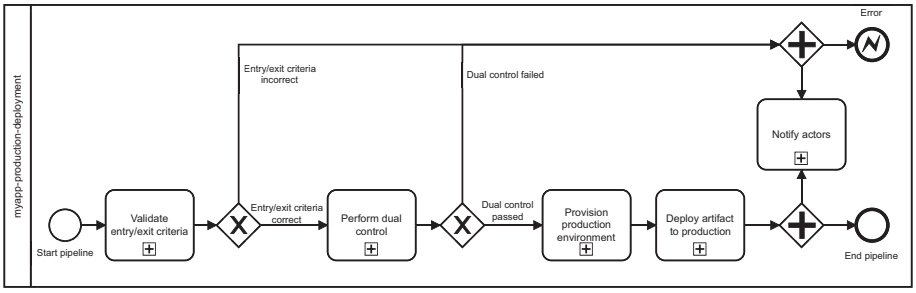


Figure 9-8. *BPMN, myapp-production-deployment*

Release Version Generation

Concerning release versioning, the semantic versioning schema is used. The release version is generated to enforce the continuity of the process, but the team does not want to rely on a specific tool. Based on the schema shown in Figure 9-9, they decide to generate the release version in the pipeline code.

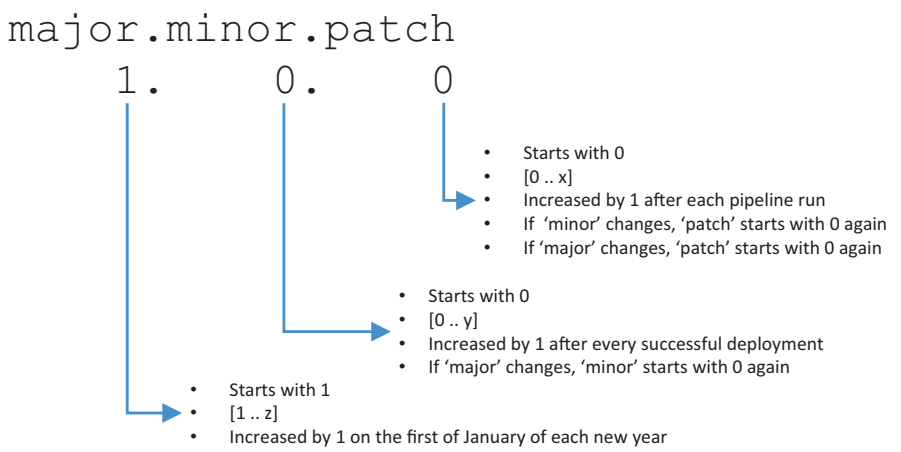

major.minor.patch
1.
0.
0
Starts with 0
[0 .. x]
Increased by 1 after each pipeline run
If 'minor' changes, 'patch' starts with 0 again
If 'major' changes, 'patch' starts with 0 again
Starts with 0
[0 .. y]
Increased by 1 after every successful deployment
If 'major' changes, 'minor' starts with 0 again
Starts with 1
[1 .. z]
Increased by 1 on the first of January of each new year

Figure 9-9. *Release version generation*

Because a biweekly release schedule is chosen, the "major" part of the version loses its value because there aren't any major releases anymore. It has been decided to increment the major every year, starting on the first of January. The minor part is incremented after every successful deployment to production and resets to zero again if the major part is incremented. The patch part of the version increments after every run of the build pipeline 2. The patch is reset to zero if the minor part changes.[3]

Pipeline Development

Before the Azure DevOps pipeline code can be used, various preparations must be made, starting with the creation of the two Azure DevOps projects. As shown in Figure 9-10, the projects are created in the Azure DevOps organization called *mycorp-com*. The projects are called *MyApp* and *MyApp-test*.

[3] If the major part is incremented, the minor is reset to zero as a result, so the patch is also reset to zero.

MyApp is the main Azure DevOps project. This is where the team develops all application and infrastructure code. The MyApp-test project is a cloned version of the MyApp project. Development and testing of pipelines happen in the MyApp-test project, so the rest of the team is not disturbed by pipeline tests. The pipeline code is merged with the code in the MyApp project after each pipeline feature is finished.

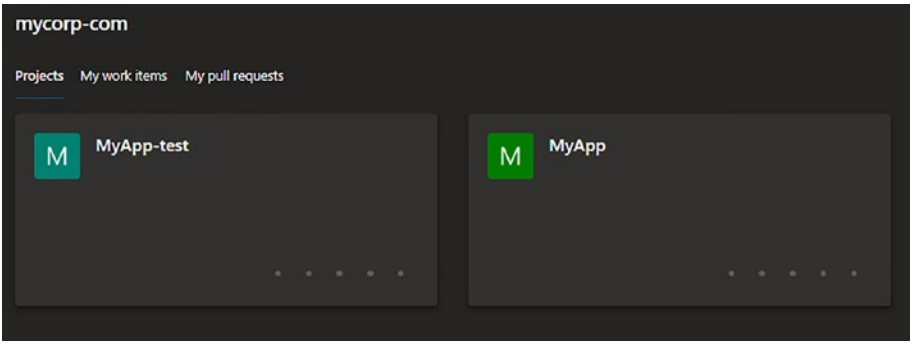


Figure 9-10. *Azure DevOps projects*

The AWSome team was so nice to share their code and their configuration, and they encourage you to use it and discover what they developed. The code provided for this book must be imported into the myapp Git repository in the MyApp project and cloned in the MyApp-test project. The preparation activities listed in this chapter apply to both projects.

Code Repository

Both Azure DevOps projects consist of three Git repositories. Two of these repositories contain scanning tools used in the pipelines. The tools Whispers and Lambdaguard are cloned from GitHub (`https://github.com/Skyscanner`) into a local repository in the Azure DevOps project to limit dependencies on Internet sources as much as possible. In addition,

these tools can also be prescanned for vulnerabilities themselves, before they are used. The third repository contains the imported code of myapp. See Figure 9-11.

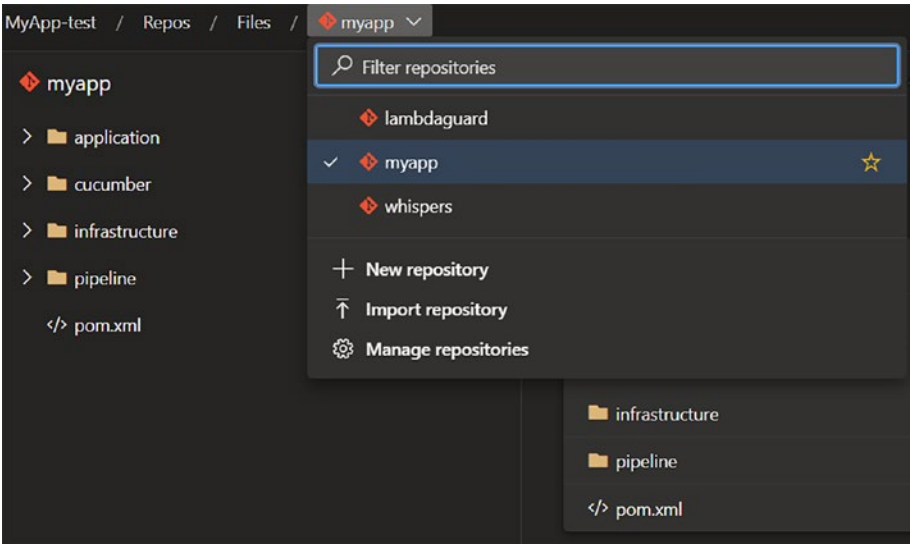


Figure 9-11. *Git repositories in Azure DevOps*

The code repository of myapp consists of a few directories.

- *application*: The healthcheck app is an AWS Lambda called *myLambda*, represented by the file `MyHandler.java`. The created healthcheck artifact is called `application*shaded.jar`.
- *cucumber*: Testing is still rudimentary, but the structure is already present in the repository. The `cucumber` directory contains a feature file called `mylambda.feature`, with just one test. The test invokes the running *myLambda* in AWS.
- *infrastructure*: The CDK code in this directory creates the lambda in AWS and installs the compiled `MyHandler.java` code. The created infrastructure artifact is called `infrastructure*shaded.jar`.

- *pipeline*: This directory contains two YAML files, called `pipeline.yml` and `prod-deployment.yml`, and represents the pipelines.
- *pipeline/template*: This directory contains a couple of template files used in the files `pipeline.yml` and `prod-deployment.yml`.
    - *deploy.yml*: This deploys the infrastructure and application JAR files to the AWS account. The AWS account and region are represented by variables, configured in a variable group (either the test or prod variable group).
    - *derive-release-version.yml:* This is a utility template to construct the release version, based on major, minor, and path parameters.
    - *download-artifacts.yml*: This downloads the build artifacts, based on a release version tag.
    - *install-tools.yml*: This installs the tools needed to deploy to AWS.
    - *provision-infra.yml*: This bootstraps the AWS account. Deploying artifacts using CDK requires some infrastructure resources, such as an S3 bucket in which the artifacts are stored.
    - *Stage-completed.yml*: As soon as a certain stage is completed (successfully), this template is called. It creates a "stage completed" file with the name of the stage. This is used to determine which QA stages of a release artifact are executed.

- *update-minor.yml*: To meet the requirements of generating the release versions, some additional code is needed. This file contains the code to update the variable group `semver`. This variable group contains the variable called `minor`, which is incremented using an Azure DevOps API.

Figure 9-12 shows the myapp repository.

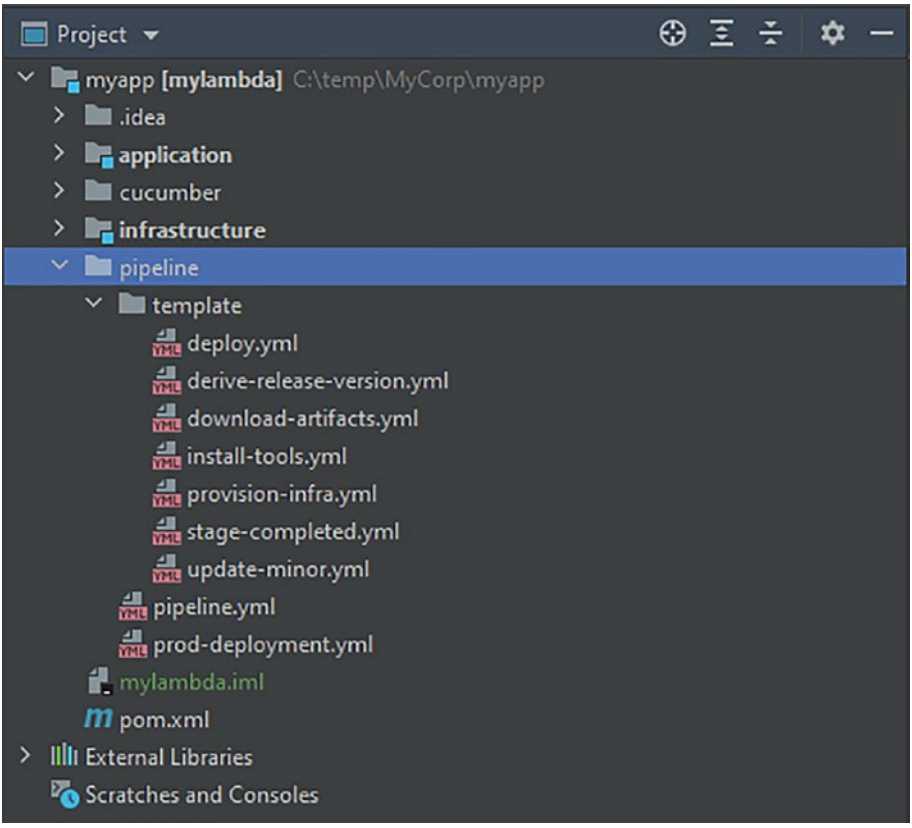


Figure 9-12. *Git repository of myapp*

Pipeline Creation

In the design phase, three logical pipelines were defined and translated into two BPMN models. The BPMN models, *myapp-pipeline* and *myapp-production-deployment*, map to two technical pipelines with the same names, as depicted by the schema in Table 9-2.

Table 9-2. *Mapping of Logical Pipelines to Technical Pipelines*

Logical Pipeline	Technical Pipeline Name	Implemented by YAML File
Pipeline, associated with the feature branch	`myapp-pipeline`	`pipeline.yml`
Primary pipeline, associated with the main branch	`myapp-pipeline`	`pipeline.yml`
Production deployment pipeline, associated with the main branch	`prod-deployment.yml`	`prod-deployment.yml`

We're assuming that the pipelines *myapp-pipeline* and *myapp-production-deployment* have been created in Azure DevOps and we are referencing the appropriate YAML files. Figure 9-13 shows the two pipelines.

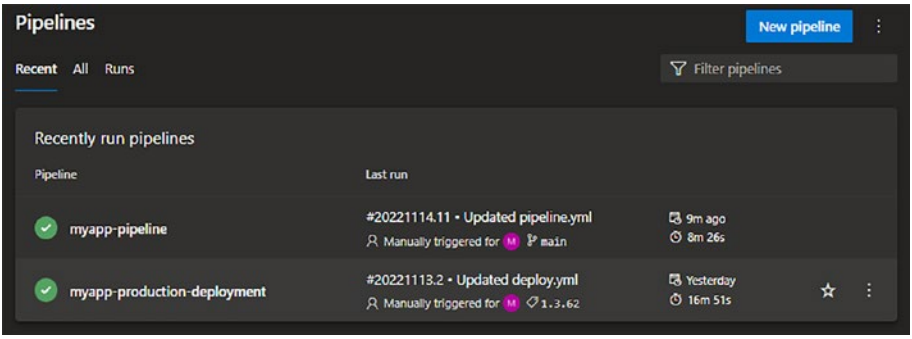


Figure 9-13. *Pipelines myapp-pipeline and myapp-production-deployment*

Configure Variable Groups

Azure DevOps has a feature called *variable groups*. This feature can be found in the main menu item on the left of the window and is called *Library*. The pipelines make use of variables defined in variable groups. Figure 9-14 gives an overview of the four variable groups that are used: *generic*, *semver*, *test*, and *prod*. Each of the latter two variable groups contain variables associated with the AWS test and production accounts.

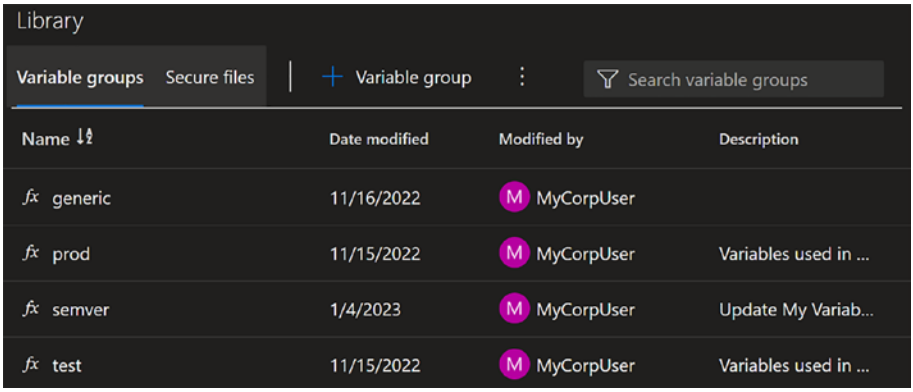


Figure 9-14. *Variable groups, overview*

Table 9-3 through Table 9-6 show the configuration of the four variable groups.

Table 9-3. *Variable Group: generic*

Name	Value	Additional Information
azdo-user	myapp@mycorp.com	
cdk-version	2.46.0	
myapp-email	myapp@mycorp.com	
nodejs-version	16.15.1	
personal-access-token	********	This is a generated personal access token (PAT); you need to generate one yourself in Azure DevOps and add it here.
pipeline-id	2	This is the pipeline ID of pipeline *myapp-pipeline*. This value can be different in your case.
project	MyApp	The value is *MyApp-test* for the test project.
rest-api-vg	`https://dev.azure.com/mycorp-com/MyApp /_apis/distributedtask/variablegroups/4?api-version=5.0-preview.1`	The Azure DevOps API to update the `semver` variable group. Note that the project in this URL is *MyApp-test* for the test project. The value 4 in this URL applies to the *semver* variable group ID. This value may be different in your situation.

(*continued*)

Table 9-3. (*continued*)

Name	Value	Additional Information
service-connection-sonarcloud	ServiceConnectionSonarCloud	The value is one string and represents the service connection (to be created).
start-year-minus-one	2022	The year before the app is released. This value is used to derive the major part of a release version.

Table 9-4. *Variable Group, semver*

Name	Value	Additional Information
last-update-year	2023	Used to determine the year of the previous release version.
minor	0	Starts with zero, but is updated after every deployment to production.

Table 9-5. *Variable Group, test (Represents the AWS Test Environment)*

Name	Value	Additional information
aws-account	497562947267	Use your AWS account if you want to try it yourself.
aws-region	us-east-1	And the region of your AWS account.
service-connection-aws-account	Service ConnectionAWS Test-497562947267	Use your own AWS service connection if you want to try it yourself.

Table 9-6. *Variable Group, prod (Represents the AWS Production Environment)*

Name	Value	Additional Information
aws-account	486439332092	In the case of Azure DevOps project *MyApp-test*, the value *497562947267* is used.
aws-region	us-east-1	
service-connection-aws-account	Service ConnectionAWS Prod-486439332092	In the case of Azure DevOps project *MyApp-test*, the value *Service ConnectionAWSTest-497562947267* is used.

Note The Azure DevOps project *MyApp-test* also contains a variable group called *prod*, but the variables in this group must refer to the AWS test account. This is only to test the *myapp-production-deployment* pipeline and not to deploy the application to the AWS production account.

See Figures 9-15 through Figure 9-18.

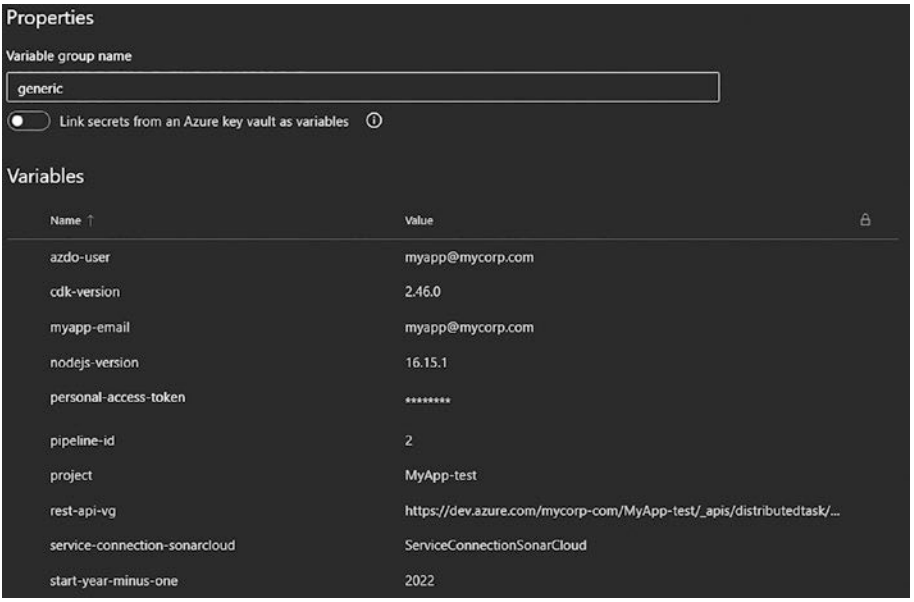


Figure 9-15. *Variable group generic*

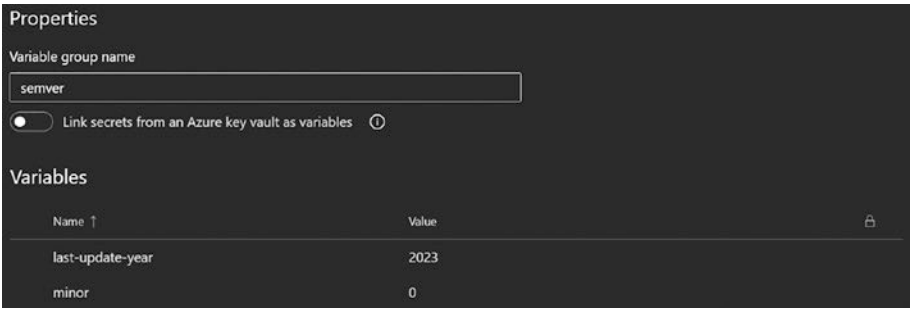


Figure 9-16. *Variable group semver*

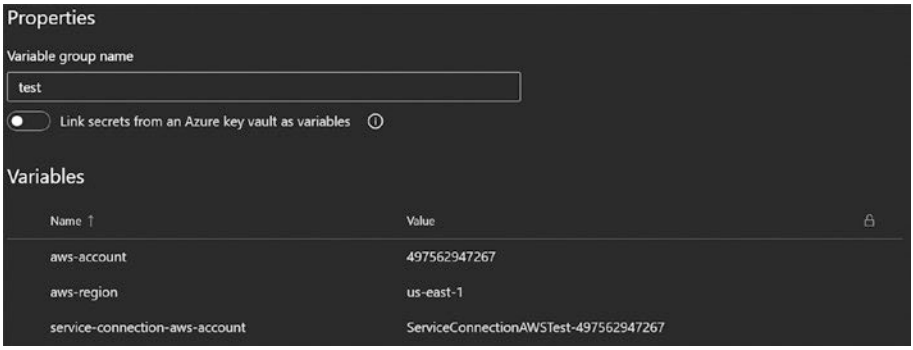


Figure 9-17. *Variable group test*

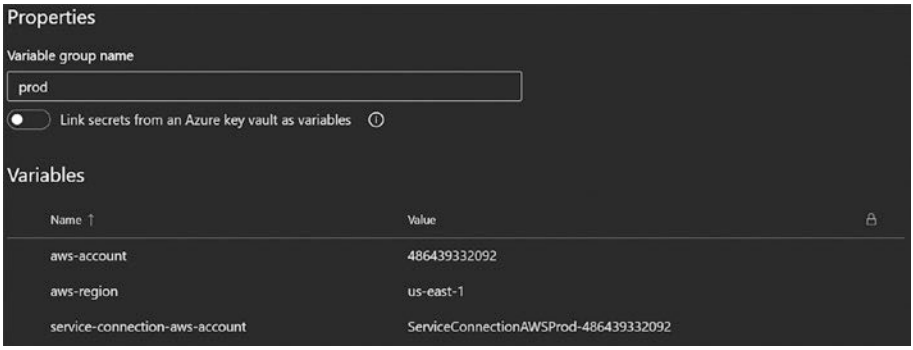


Figure 9-18. *Variable group prod*

Configure Service Connections

Azure DevOps makes use of service connections to connect to the AWS target environments and SonarCloud. The extensions for AWS and SonarCloud can be downloaded from the Internet in the Azure DevOps marketplace. See Figure 9-19.

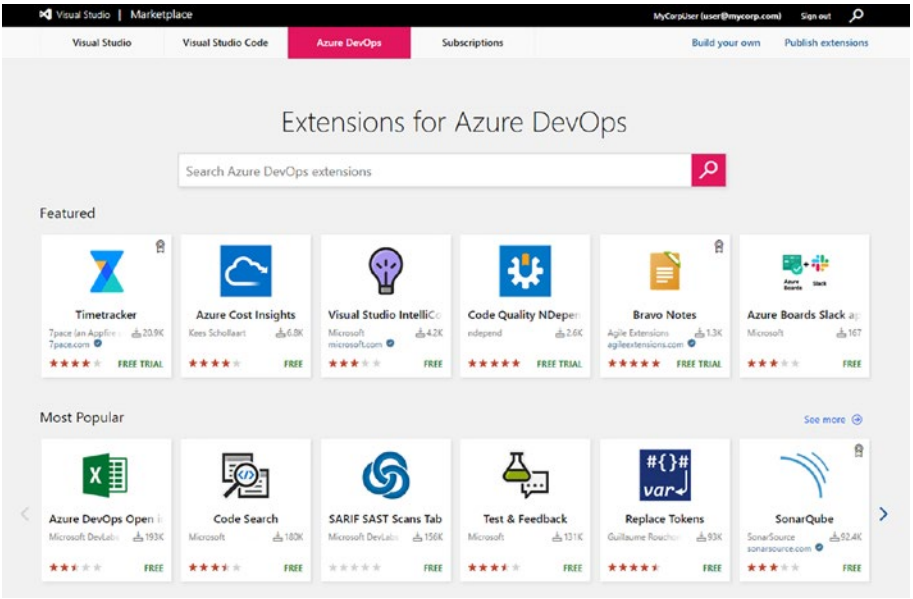


Figure 9-19. *Azure DevOps marketplace*

The pipelines of the AWSome team make use of the following marketplace extensions:

- AWS Toolkit for Azure DevOps
- SonarCloud
- SonarCloud build breaker

If these extensions are installed in your Azure DevOps organization, they can be used to create AWS and SonarCloud service connections.

Figure 9-20 shows an overview of the three service connections. No detailed step-by-step description is given on how to set up a service connection, but the service connections require at least the information in Table 9-7 and Table 9-8.

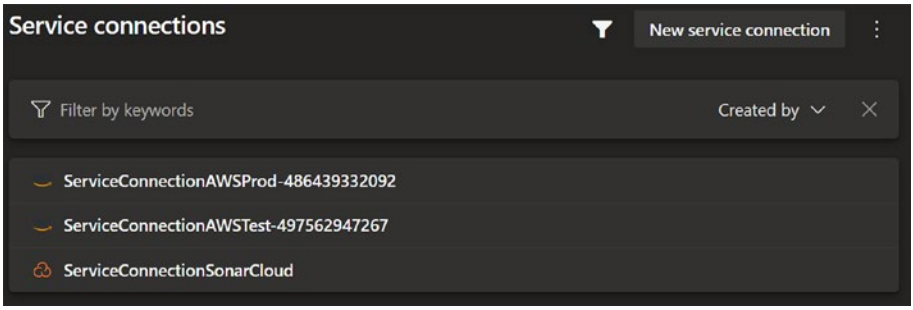


Figure 9-20. *Service connections overview*

Table 9-7. *Service Connection AWS, test and prod*

AWS Service Connection (see Figure 9-21; left image)	
Access Key ID	Acquired from the AWS account
Secret Access Key	Acquired from the AWS account
Service connection name	Either ServiceConnectionAWSTest-497562947267 (for test) or ServiceConnectionAWSProd-486439332092 (for production)

Table 9-8. *Service Connection Sonar Cloud*

SonarCloud Service Connection (see Figure 9-21; right image)	
SonarCloud Token	Acquired from SonarCloud after registration of the Azure DevOps project.
Service connection name	ServiceConnectionSonarCloud

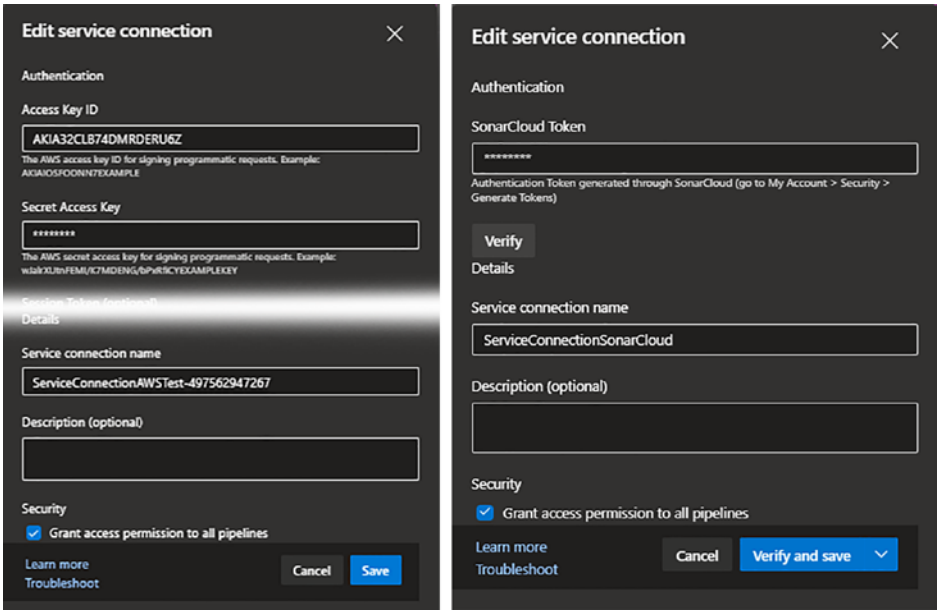


Figure 9-21. *Service connections*

Test

This section does not go into much detail on testing pipelines. Development and testing are done in a separate Azure DevOps project, so from a pipeline testing point of view, some measures are taken to optimize pipeline testing.

Executing pipeline *myapp-pipeline* results in images similar to Figure 9-22 and Figure 9-23. The first figure represents the stages if the pipeline is associated with a feature branch. The next figure shows the stages associated with the main branch. As shown in Figure 9-23, release version 1.0.3 is created. All the stages are passed, and the application is deployed to the AWS test environment.

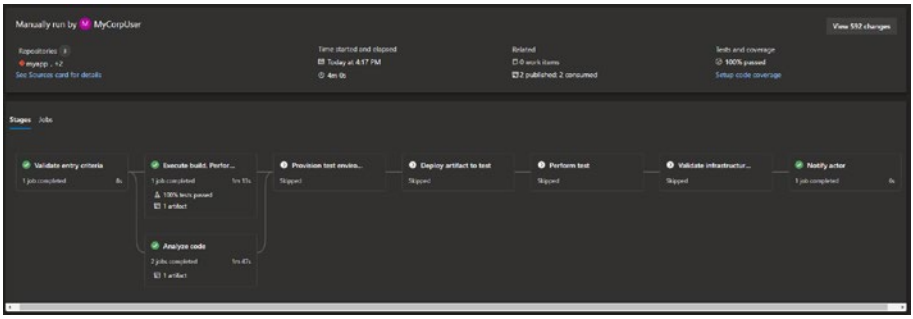


Figure 9-22. *Run of pipeline myapp-pipeline associated with a feature branch*

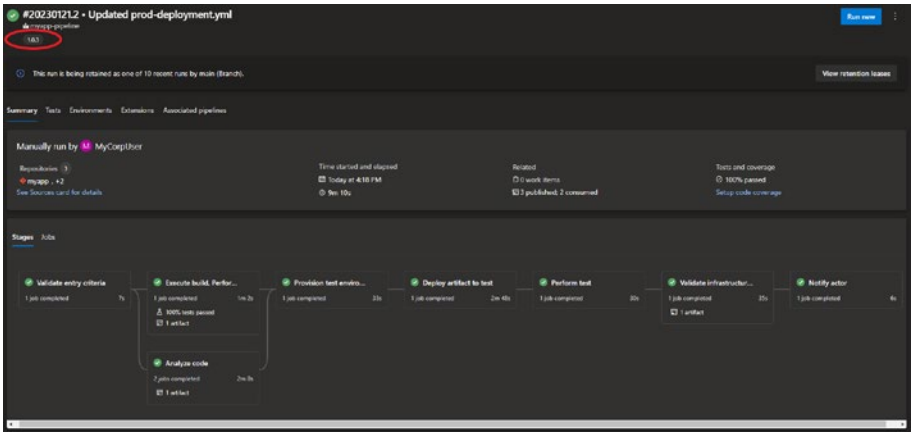


Figure 9-23. *Run of pipeline myapp-pipeline version 1.0.3 (main branch)*

Let's zoom in on some of the stages. The *Analyze code* stage consists of a SonarCloud scan with a build breaker and a Whispers scan, represented respectively by Figure 9-24 and Listing 9-1.

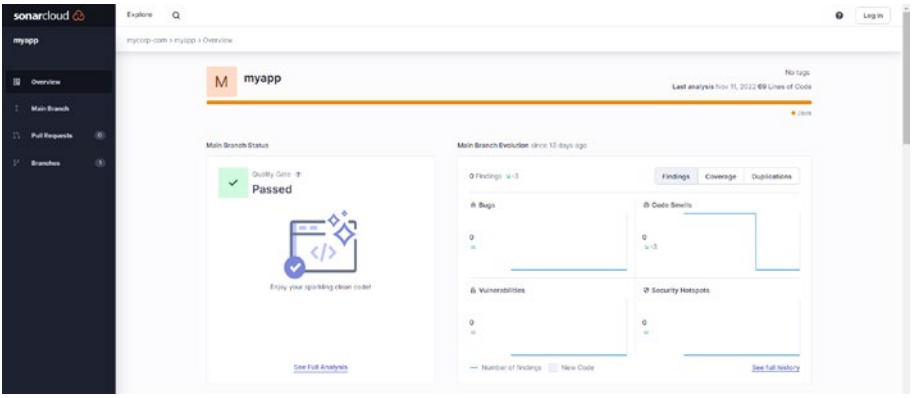


Figure 9-24. *SonarCloud scan*

Listing 9-1. Log of the Whispers Scan

```
Starting: Scan to find hardcoded credentials and dangerous
functions
==============================================================
Task         : Command line
Description  : Run a command line script using Bash on Linux
               and macOS and cmd.exe on Windows
Version      : 2.201.1
Author       : Microsoft Corporation
Help         : https://docs.microsoft.com/azure/devops/
               pipelines/tasks/utility/command-line
==============================================================
Generating script.
================== Starting Command Output ==================
/usr/bin/bash --noprofile --norc /home/vsts/work/_
temp/2a18c6d1-7a8d-4bfc-af91-bea2556185b6.sh
pip3 install -e .
............
```

```
Installing collected packages: rapidfuzz, Levenshtein, python-
levenshtein, soupsieve, beautifulsoup4, lazy-object-proxy,
wrapt, typing-extensions, astroid, jproperties, luhn, lxml,
whispers
  Running setup.py develop for whispers
Successfully installed Levenshtein-0.20.8 astroid-2.12.12
beautifulsoup4-4.11.1 jproperties-2.1.1 lazy-object-
proxy-1.8.0 luhn-0.2.0 lxml-4.9.1 python-levenshtein-0.20.8
rapidfuzz-2.13.2 soupsieve-2.3.2.post1 typing-extensions-4.4.0
whispers wrapt-1.14.1
Scan myapp
Finishing: Scan to find hardcoded credentials and dangerous
functions
```

Both scans show that everything is fine. The build passes the SonarCloud quality gate and the Whispers scan looks fine (no hard-coded secrets).

The *Perform test* stage contains a test task invoking a Cucumber test. The test is still simple and covers only one test, defined in the `mylambda.feature` file shown in Listing 9-2.

Listing 9-2. Feature File

```
Feature: Is the response ok?
  Sending a request should return a valid response

  Scenario: Validate status of the response after handling
            an event
    Given myLambda is running
    When I send a valid request
    Then I should get status "\"200 OK\""
```

This results in the output shown in Listing 9-3.

Listing 9-3. Log of the Cucumber Test

```
┌──────────────────────────────────────────────────────────┐
│ Share your Cucumber Report with your team at             │
│ https://reports.cucumber.io                              │
│ Activate publishing with one of the following:           │
│                                                          │
│ src/test/resources/cucumber.properties:                  │
│         cucumber.publish.enabled=true                    │
│ src/test/resources/junit-platform.properties:            │
│          cucumber.publish.enabled=true                   │
│ Environment variable:     CUCUMBER_PUBLISH_ENABLED=true  │
│ JUnit:                   @CucumberOptions(publish = true)│
│                                                          │
│ More information at https://cucumber.io/docs/            │
│ cucumber/environment-variables/                          │
│                                                          │
│ Disable this message with one of the following:          │
│                                                          │
│ src/test/resources/cucumber.properties:                  │
│         cucumber.publish.quiet=true                      │
│ src/test/resources/junit-platform.properties:            │
│          cucumber.publish.quiet=true                     │
└──────────────────────────────────────────────────────────┘
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
Time elapsed: 2.757 s - in mylambda.RunCucumberTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time:  12.905 s
[INFO] Finished at: 2022-11-11T18:12:29Z
[INFO] ------------------------------------------------------------
```

The *Validate infrastructure compliance* stage contains a validation of the *myLambda* configuration in the AWS account using Lambdaguard, as shown in the log (Listing 9-4).

Listing 9-4. Log of Lambdaguard

```
         `.::////::.`
      ./osssssoosssso/.
    -osss/-`      .-/ssso-
  `osso-  .++++:      -osso`
 `oss/    .//oss-       /sss`
 +ss+        -sss.       /sso
.sss`       .sssso`      `sss.   LambdaGuard v2.4.3
-sso       :ssooss+       oss-
.sss`     /ss+``oss/     `sss.
 +ss+   `oss/   .sss///  /sso
 `oss/`.oso-     -ssso+./sso`
  `+sso:           .`  -oss+`
    -osss+-.`    `.-+ssso-
      ./ossssssssssso/.
         `.-:////:-.`

Loading regions (us-east-1)
Loading identity
          UserId..... AKIA32CLB74DMRDERU6Z
          Account..... 497562947267
```

```
          Arn..... arn:aws:iam::497562947267:user/
          azuredevops

[ 1/1 ] myLambda
          Lambdas........ 1
          Security....... 2
          Triggers....... 1
          Resources...... 0
          Layers......... 0
          Runtimes....... 1
          Regions........ 1
          Report......... ./mylambda-report/report.html
          Log............ ./mylambda-report/lambdaguard.log

Finishing: Intall Lambdaguard and validate myLambda in AWS
```

The report is published as part of this pipeline and can be downloaded. Some attention is needed because the AWSLambdaBasicExecutionRole has more privileges than needed; these privileges need to be restricted. See Figure 9-25.

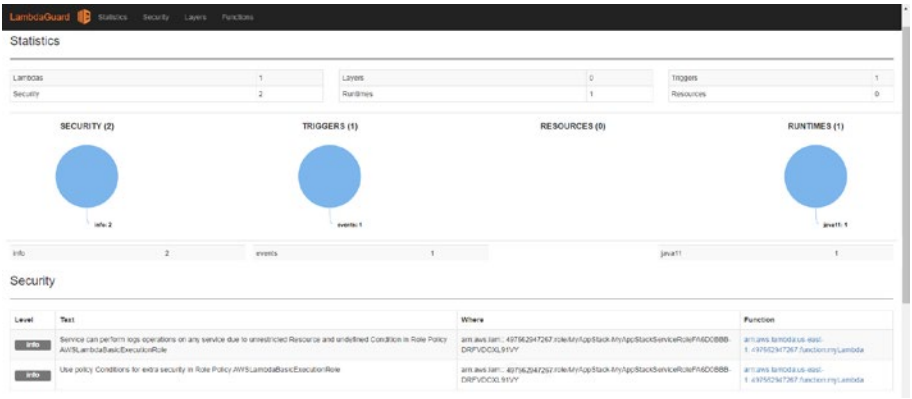


Figure 9-25. *Lambdaguard report*

Integrity of Artifacts

The security requirement "Only build and deploy artifacts using a pipeline" states that the integrity of the artifact must be guaranteed, from building the artifact to running the artifact. A simple measure is applied to meet this requirement. The first step in this process is to visualize that the integrity remains the same over all stages in the process. This is done by creating an SHA256 hash of the built artifact. If the hash of the lambda running in the AWS target environment is the same as the hash of the artifact in the pipeline(s), there is high confidence that it is the same artifact. Generating the SHA256 hash is included in the files `pipeline.yml` and `template/deploy.yml`. Pipelines *myapp-pipeline* and *myapp-production-deployment* both print the hash in the log, as shown in Listing 9-5 and Listing 9-6.

Listing 9-5. Log of the build, myapp-pipeline

```
Starting: Calculate SHA256 checksum of the application jar file
==============================================================
Task         : Command line
Description  : Run a command line script using Bash on Linux
               and macOS and cmd.exe on Windows
Version      : 2.201.1
Author       : Microsoft Corporation
Help         : https://docs.microsoft.com/azure/devops/
               pipelines/tasks/utility/command-line
==============================================================
Generating script.
================== Starting Command Output ===================
/usr/bin/bash --noprofile --norc /home/vsts/work/_temp/830814d
3-13e0-40ee-9c2b-9eb5ea3ca74d.sh
```

```
SHA256 checksum of /home/vsts/work/1/s/application/target/
application-1.3.62-shaded.jar
58c3de378ff9016bdf0c71781134672f1e4efa8801d46ef99427d160
afad3a10  /home/vsts/work/1/s/application/target/
application-1.3.62-shaded.jar
Finishing: Calculate SHA256 checksum of the application
jar file
```

Listing 9-6. Log of the Deployment, myapp-production-deployment

```
Starting: Deploy to AWS
==============================================================
Task         : AWS Shell Script
Description  : Run a shell script using Bash with AWS
               credentials as environment variables
Version      : 1.13.0
Author       : Amazon Web Services
Help         : Runs a shell script in Bash, setting AWS
credentials and region information into the shell environment
using the standard environment keys _AWS_ACCESS_KEY_ID_, _AWS_
SECRET_ACCESS_KEY_, _AWS_SESSION_TOKEN_ and _AWS_REGION_.

More information on this task can be found in the [task
reference](https://docs.aws.amazon.com/vsts/latest/userguide/
awsshell.html).

####Task Permissions
Permissions for this task to call AWS service APIs depend on
the activities in the supplied script.
==============================================================
Configuring credentials for task
```

```
...configuring AWS credentials from service endpoint 'b43bf
786-1c0c-45f7-9f98-fd31a2d01b0a'
...endpoint defines standard access/secret key credentials
Configuring region for task
...configured to use region us-east-1, defined in task.
/usr/bin/bash /home/vsts/work/_temp/awsshellscript_2012.sh
artifacts/infrastructure-1.3.62-shaded.jar'
Infrastructure artifact name and path: /home/vsts/work/1/
myapp-artifacts/infrastructure-1.3.62-shaded.jar
Application artifact name and path: /home/vsts/work/1/
myapp-artifacts/application-1.3.62-shaded.jar
Version to deploy: 1.3.62
SHA256 checksum of /home/vsts/work/1/myapp-artifacts/
application-1.3.62-shaded.jar
58c3de378ff9016bdf0c71781134672f1e4efa8801d46ef99427
d160afad3a10  /home/vsts/work/1/myapp-artifacts/
application-1.3.62-shaded.jar
```

The hash of *myLambda*, deployed to AWS, is displayed in the AWS console, as depicted in Figure 9-26. This hash, `WMPeN4/5AWvfDHF4ETRnLx5O+ogB1G75lCfRYK+tOhA=`, is in Base64 format, though. It needs to be converted to a hash in Hex format to compare it (you can use https://base64.guru/converter/decode/hex). This results in the hash `58c3de378ff9016bdf0c71781134672f1e4efa8801d46ef99427d160afad3a10`, indicating that *myLambda*, running in AWS, is the same as built and deployed using the pipelines.

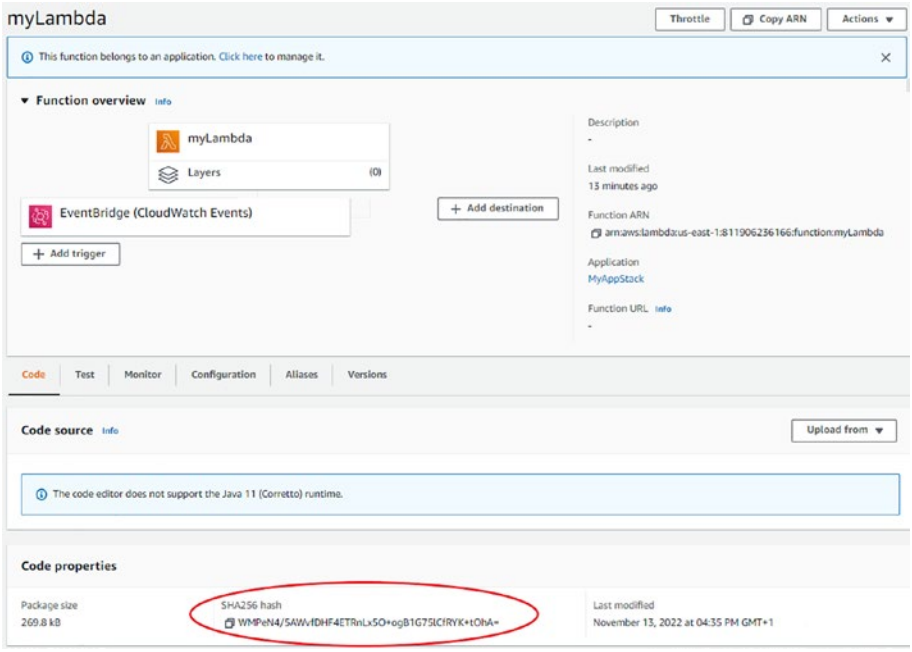


Figure 9-26. *AWS console, myLambda SHA256 hash*

This is a rudimentary integrity check. The integrity is not monitored throughout the life cycle of the Lambda. If someone were to replace the Lambda, it can be done without being noticed. Lambda code signing and monitoring are additional measures to guarantee integrity over the life cycle of *myLambda*.

Performance and Acceptance Pipelines

The performance of *myapp-pipeline* varies, depending on the time of the day. The overall execution time approximately lies between 7 and 15 minutes. Caching is enabled, and code analysis is executed in parallel. By looking at the individual execution times of each stage, there isn't much to improve on without doing in-depth research or switching to self-hosted agents. The performance of the *myapp-production-deployment* pipeline is

a lot faster. The wait time before a dual control is performed is many times greater than the actual execution time of the stages. With these numbers and the fact that the outcome of all stages looks good, the AWSome team approves the pipeline, which can be implemented in the MyApp project.

Implementation

Implementation means that the pipeline developed and tested in the MyApp-test project is pushed to the MyApp project. The first increment does not cover all requirements, and some mitigating actions are applied. The team puts work items on the backlog that need to be implemented in the next couple of iterations. Here is a selection from their backlog:

- *Workitem 1*: The requirement "Resources associated with a release cannot be deleted" is not implemented. This is put on the backlog. Retaining pipelines for a long time can be automated, using the Leases API of Azure DevOps, which sets the retention time of a pipeline to "forever" after a deployment to production.

  Mitigating action: As a contingency measure, the retention times are increased (see Figure 9-27), and releases deployed to production are retained manually (see Figure 9-28).

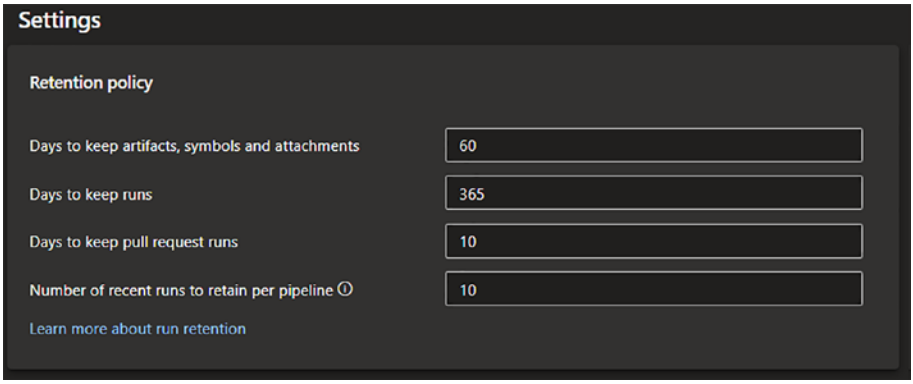


Figure 9-27. *Retention times*

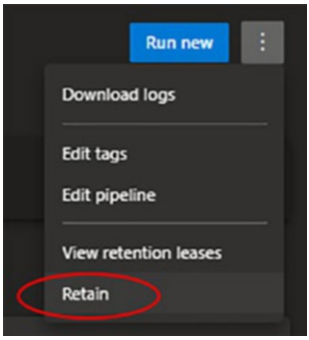


Figure 9-28. *Retain the pipeline run manually*

- *Workitem 2*: During the build, libraries are directly retrieved from the central Maven repository (`https://repo.maven.apache.org/maven2/`). As the first line of defense, a proxy must be set up, so the libraries are retrieved using this proxy. In addition, a Nexus repository will be installed to store the libraries locally within the organizations' boundaries.

- *Workitem 3*: The Whispers task does not break the build, irrespective of the result. Add a check to break the build if a vulnerability is detected.

  Mitigating action: Validate the result manually.

- *Workitem 4*: The AWSLambdaBasicExecutionRole used to execute *myLambda* is not restrictive enough. Create a new role for this Lambda, and apply the principle of least privilege access.

- *Workitem 5*: The deployment strategy should change from the re-create to canary deployment strategy using AWS CodeDeploy.

Configure the Azure DevOps Prod Environment and Dual Control

If nothing is specified, Azure DevOps automatically creates an Azure DevOps environment when it encounters an `environment` setting in the pipeline. In the Azure DevOps test environment (MyApps-test), both the test and prod environments are automatically created when the pipeline runs. However, as part of the pipeline implementation, the prod environment needs to be configured in the MyApp project to allow dual control.

As a result of the requirement "Refine access by setting permissions for a user or group," the AWSome team is created in the permissions configuration of project MyApps, as shown in Figure 9-29. All team members are added. In addition, a new group is created, called Product Owner. Figure 9-29 shows the Product Owner group to which Emma and Vinod are assigned. This group is used in the dual control configuration, so only Emma and Vinod are allowed to approve a deployment to production.

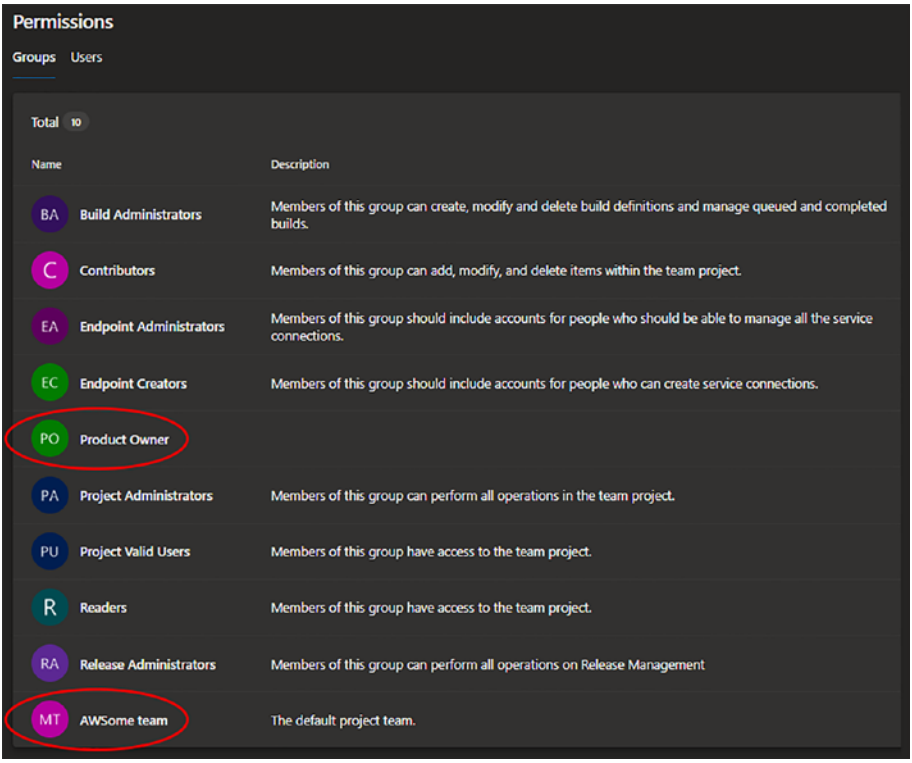

Permissions
Groups
Users
Total
10
Name
Description
BA
Build Administrators
Members of this group can create, modify and delete build definitions and manage queued and completed builds.
C
Contributors
Members of this group can add, modify, and delete items within the team project.
EA
Endpoint Administrators
Members of this group should include accounts for people who should be able to manage all the service connections.
EC
Endpoint Creators
Members of this group should include accounts for people who can create service connections.
PO
Product Owner
PA
Project Administrators
Members of this group can perform all operations in the team project.
PU
Project Valid Users
Members of this group have access to the team project.
R
Readers
Members of this group have access to the team project.
RA
Release Administrators
Members of this group can perform all operations on Release Management
MT
AWSome team
The default project team.

Figure 9-29. *Product Owner group and AWSome team*

The test and prod environments as depicted in Figure 9-30 are manually created, and an Approval is added to the prod environment, as shown in Figure 9-31. This approval implements the *Perform dual control* stage. Also note that members of the Product Owner group are not allowed to both start and approve a pipeline. This contributes to a more secure pipeline.

Figure 9-30. *Environments prod and test*

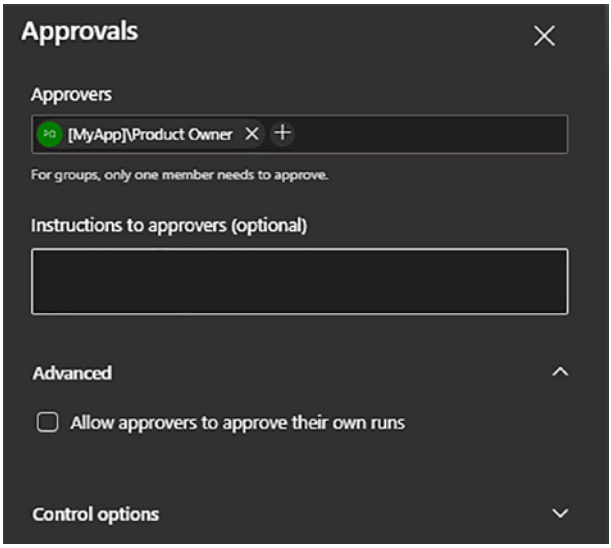


Figure 9-31. *Configuring approvals for the prod environment*

Deploy the Application to Production

The goal of the pipelines is of course to deploy the application to production. The deployment to production is performed using the *myapp-production-deployment* pipeline. This pipeline is constructed in such a way that it deploys the artifacts of a certain release, based on the release tag selected in the start dialog. This is shown in Figure 9-32. In this example, release 1.0.3 is selected and deployed.

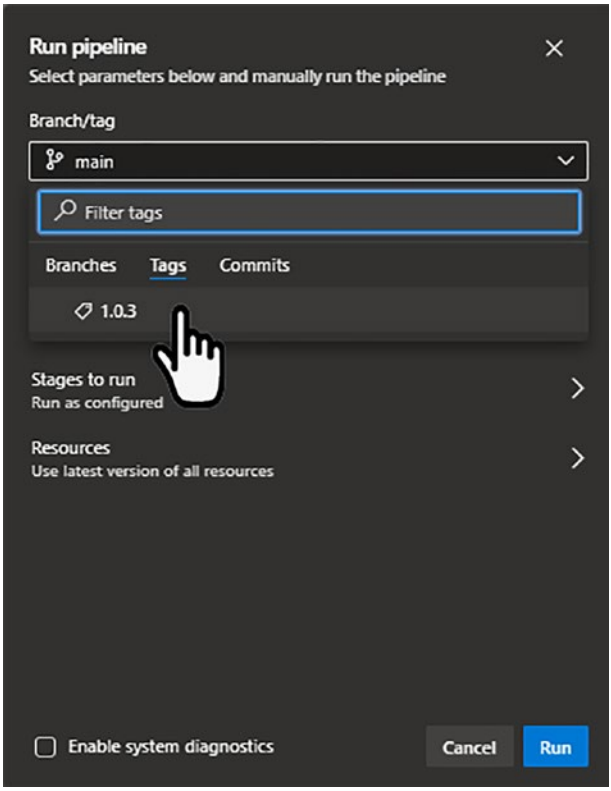


Figure 9-32. *Starting the production deployment pipeline*

As soon as the pipeline reaches the dual control step, it shows a dialog similar to Figure 9-33. Members of the Product Owner group must approve (or reject) it before the deployment to production is performed.

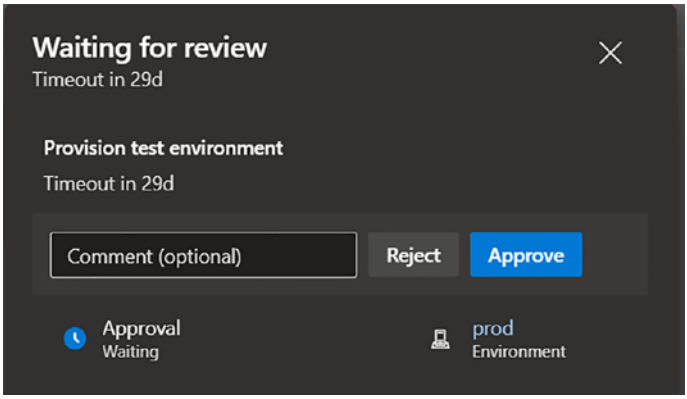


Figure 9-33. *Dual control*

The stages of the *myapp-production-deployment* pipeline are depicted in Figure 9-34. Also, take note of the fact that the pipeline run is tagged with release version 1.0.3.

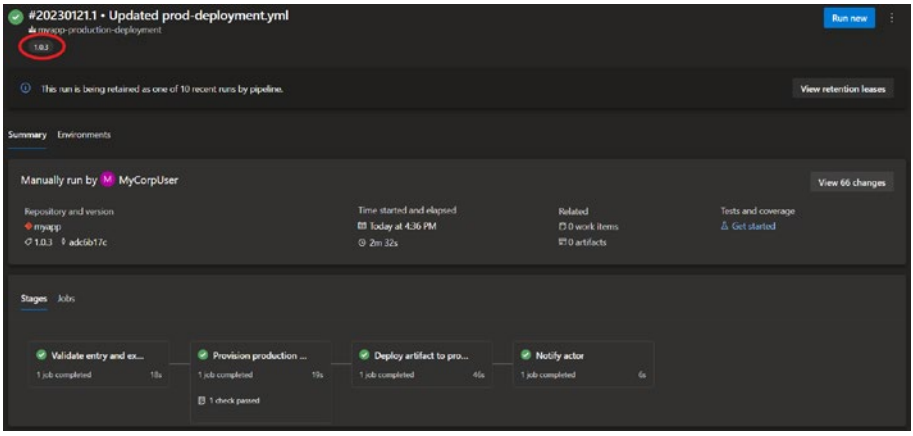


Figure 9-34. *Stages production deployment pipeline*

The result is the deployment of the artifact stack, which includes the *myLambda* resource, in the AWS production account (see Figure 9-35). Notice the presence of the release version tag in this stack. This completes the requirement "All changes are traceable/Tag everything."

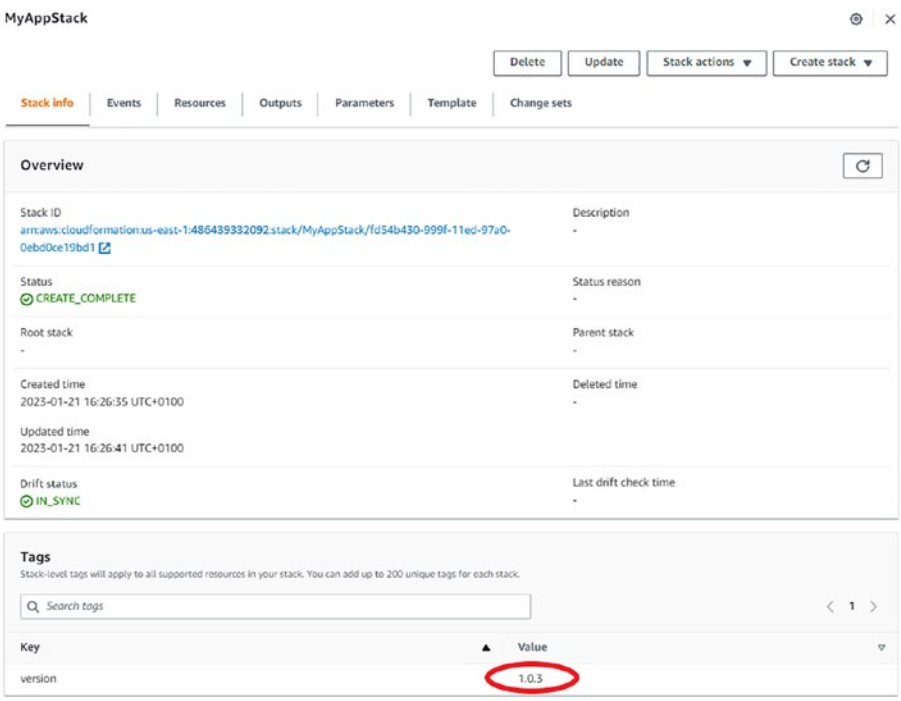


Figure 9-35. *AWS CloudFormation stack MyAppStack (including myLambda)*

To prove the working of the *myLambda* healthcheck, an excerpt of the CloudWatch log is included, which shows the log lines produced by the *myLambda* healthcheck. See Figure 9-36.

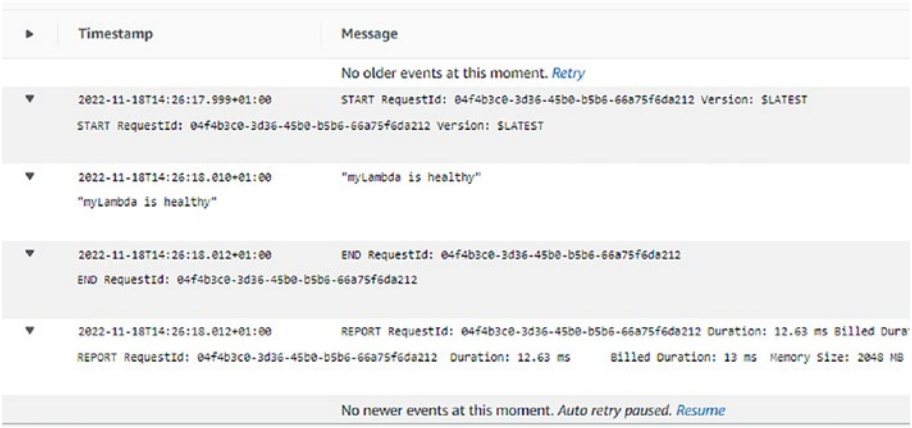

Timestamp
Message
No older events at this moment. Retry
2022-11-18T14:26:17.999+01:00
START RequestId: 04f4b3c0-3d36-45b0-b5b6-66a75f6da212 Version: $LATEST
START RequestId: 04f4b3c0-3d36-45b0-b5b6-66a75f6da212 Version: $LATEST
2022-11-18T14:26:18.010+01:00
"myLambda is healthy"
"myLambda is healthy"
2022-11-18T14:26:18.012+01:00
END RequestId: 04f4b3c0-3d36-45b0-b5b6-66a75f6da212
END RequestId: 04f4b3c0-3d36-45b0-b5b6-66a75f6da212
2022-11-18T14:26:18.012+01:00
REPORT RequestId: 04f4b3c0-3d36-45b0-b5b6-66a75f6da212 Duration: 12.63 ms Billed Dura
REPORT RequestId: 04f4b3c0-3d36-45b0-b5b6-66a75f6da212 Duration: 12.63 ms Billed Duration: 13 ms Memory Size: 2048 MB
No newer events at this moment. Auto retry paused. Resume

Figure 9-36. *AWS myLambda log*

Quality Gate

To prevent an incorrect release version from being deployed to production, an additional quality gate is added to the *myapp-production-deployment* pipeline. This quality gate prevents that release versions, for which the stages *Analyze code*, *Perform test*, and *Validate infrastructure compliance* are not executed, can be deployed to production.

The pipeline *myapp-pipeline* creates a "stage completed" file after every successful run of a particular stage. Only release versions for which the files `ANALYZE-CODE-COMPLETED`, `PERFORM-TEST-COMPLETED`, and `VALIDATE-INFRASTRUCTURE-COMPLIANCE-COMPLETED` are created and considered valid releases. The existence of these files is checked in the *Validate entry/exit criteria* stage in pipeline *myapp-production-deployment*.

Figure 9-37 shows the artifacts of *myapp-pipeline*. The three "stage completed" files are listed in the `myapp-status` folder.

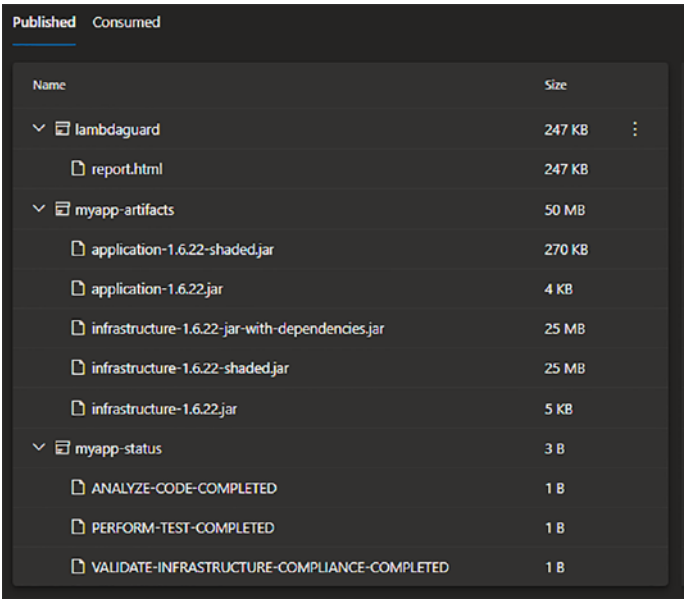


Figure 9-37. *All artifacts of myapp-pipeline*

If one of these files is not present, the *myapp-production-deployment* pipeline fails, as shown in Listing 9-7.

Listing 9-7. Log of a Failed Deployment (Noncompleted Stage in myapp-pipeline)

```
Starting: Validate whether QA stages are completed
==============================================================
Task         : Command line
Description  : Run a command line script using Bash on Linux
               and macOS and cmd.exe on Windows
Version      : 2.212.0
Author       : Microsoft Corporation
Help         : https://docs.microsoft.com/azure/devops/
               pipelines/tasks/utility/command-line
==============================================================
```

```
Generating script.
================== Starting Command Output ====================
/usr/bin/bash --noprofile --norc /home/vsts/work/_temp/
bb88fef0-b881-44a8-b3da-06cc6a165198.sh
Stage [Validate infrastructure compliance] was not executed
##[error]Bash exited with code '1'.
Finishing: Validate whether QA stages are completed
```

Summary

You learned about the following topics in this chapter:

- You learned, based on requirements, how to derive a design and how this design translates to a technical pipeline implementation, based on the approach described in the previous chapters.
- We demonstrated how the structure of the pipeline repository is set up.
- The provided Azure DevOps pipeline code and the detailed description of the Azure DevOps project configuration illustrated how to develop pipelines that meet the requirements.
- The execution of the pipeline stages, the code analysis, the infrastructure compliance results, the test results, and the output of the running app showed how the pipelines work.
- Attention is given to some specific (security) requirements, such as the integrity of artifacts and dual control.

- Implementation of the pipeline shows that it does not have to be a problem if not all requirements are implemented in the first increment, as long as this is recognized and recorded.
- Additional quality gates can be added to a pipeline to prevent deployments of release candidates that did not pass all QA tests. A simple example is given, which makes use of “stage completed” files to earmark executed QA stages.

References

[1] Cambridge Bitcoin Electricity Consumption Index
https://ccaf.io/cbeci/index

[2] Business Process Model and Notation
https://www.omg.org/spec/BPMN/2.0

[3] Cloudbees CD
https://docs.cloudbees.com/docs/cloudbees-cd/10.0/

[4] *Succeeding with Agile*
Mike Cohn
Pearson Education, 2009
EAN 9780321579362

[5] *Continuous Integration: Improving Software Quality and Reducing Risk*
Paul M. Duvall, Steve Matyas, and Andrew Glover.
Addison-Wesley, 2007
EAN 9780321336385

[6] *Continuous Delivery: Reliable Software Release through Build, Test and Deployment Automation*
Jez Humble and David Farley
Addison-Wesley, 2010
EAN 9780321601919
https://continuousdelivery.com/

[7] The Open Group IT4IT™ Reference Architecture, Version 2.1
https://pubs.opengroup.org/it4it/refarch21/

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0

[8] Enterprise CI/CD: Best Practices
Kostis Kapelonis (Codefresh)
https://codefresh.io/ebooks/enterprise-ci-cd-best-practices/

[9] *Energy Efficiency Across Programming Languages*
Rui Pereira e.o.
Universidade do Minho, Portugal
https://www.researchgate.net/publication/320436353_Energy_efficiency_across_programming_languages_how_do_energy_time_and_memory_relate

[10] Enforce Signed Software Execution Policies
https://media.defense.gov/2019/Sep/09/2002180334/-1/-1/0/Enforce%20Signed%20Software%20Execution%20Policies%20-%20Copy.pdf

[11] National Cyber Security Center
https://www.ncsc.gov.uk/

[12] National Information Assurance Partnership
https://www.niap-ccevs.org/

[13] NIST Cybersecurity Framework
https://www.nist.gov/cyberframework

[14] Rapid Release at Massive Scale: CI/CD at Facebook
https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/

[15] Continuous Deployment of Mobile Software at Facebook (Showcase)
https://research.facebook.com/publications/continuous-deployment-of-mobile-software-at-facebook-showcase/

[16] *Workflow Patterns: The Definite Guide*
Nick Russel, Wil M.P. van der Aalst, Arthur H.M. ter Hofstede
The MIT Press, 2016
ISBN 978-0-262-02982-7

[17] Gerrit Code Review
https://www.gerritcodereview.com/

[18] Gitflow Workflow
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[19] Semantic Versioning
https://semver.org/

[20] Microsoft Teams: Webhooks and Connectors
https://docs.microsoft.com/en-us/microsoftteams/platform/webhooks-and-connectors/what-are-webhooks-and-connectors

[21] *Design Patterns: Elements of Reusable Object-Oriented Software*
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang of Four)
Addison-Wesley Professional, 1994
ISBN 978-0201633610

[22] In-toto
https://in-toto.io/
Argos Notary
https://www.argosnotary.com/

[23] GitLab Docs
https://docs.gitlab.com/

[24] Feature Management Systems
https://www.getunleash.io/
https://launchdarkly.com/

[25] ISO 25010
https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[26] NIST Framework for Improving Critical Infrastructure Cybersecurity
https://www.nist.gov/publications/framework-improving-critical-infrastructure-cybersecurity-version-11

[27] Patterns for Managing Source Code Branches
https://martinfowler.com/articles/branching-patterns.html

[28] Continuous Delivery
YouTube subscription Dave Farley
https://www.youtube.com/c/ContinuousDelivery

[29] The Principles of Sustainable Green Software Engineering
Asim Hussain, Green Cloud Advocacy Lead at Microsoft
https://principles.green/

[30] Azure Sustainability
https://azure.microsoft.com/en-us/explore/global-infrastructure/sustainability/#overview

[31] AWS Energy Transition
https://aws.amazon.com/energy/sustainability/

[32] Open Policy Agent
https://www.openpolicyagent.org/

[33] *Microsoft Azure Essentials Azure Automation*
Michael McKeown
Microsoft Press, 2015
ISBN: 978-0-7456-9815-4

[34] Pyrsia
https://pyrsia.io/

[35] Continuous Delivery Foundation
https://cd.foundation/

[36] National Institute for the Software Industry
https://nisi.nl/

[37] Secret Management Tools
https://github.com/mozilla/sops
https://github.com/StackExchange/blackbox

Index

A

A/B testing, 78, 180–183, 253
Acceptance tests, 140, 288, 307, 362
Actors, 102, 103
Alerts, 72, 73, 352–356
Analyze code, 22, 92–93, 195–200
API tests, 141, 151, 155
Application life-cycle management (ALM), 36, 38, 39, 44, 51, 62, 90, 113, 130, 153, 209, 244, 250, 317
  auto-cancel, 151, 230–231
  tools, 16, 39
application*shaded.jar, 372
Artifacts, 51, 94, 101, 125, 326
Artificial intelligence (AI), 15, 31, 74, 90
Atlassian's Bamboo, 210
Automated process, 91, 187, 200, 318, 347
Automated security tests (IAST/DAST), 153, 157
Automated tests, 8, 30, 37, 66, 144, 155, 185, 314
AWS Key Management Services, 251
AWSLambdaBasicExecutionRole, 390, 397
AWS Secrets Manager, 251
Azure DevOps, 21, 234, 360, 365
  branching strategy, 367
  context diagram, 366
  environment, 290, 365
  exit criteria stage, 368
  myapp-pipeline, 368
  release strategy, 367
  release version generation, 369, 370
Azure Key Vault, 251
Azure resource manager (ARM), 5

B

Binary repository, 22, 24, 52, 60, 94, 135, 325
Blue/green deployment, 170, 173, 175, 181
BPMN 2.0, 80
Branching strategy, 3, 6, 30, 33, 105, 158, 271, 350, 361, 367
Business continuity, 63
Business organization, 30, 32, 40, 68, 75, 165, 305

© Henry van Merode 2023
H. van Merode, *Continuous Integration (CI) and Continuous Delivery (CD)*,
https://doi.org/10.1007/978-1-4842-9228-0

Business process model and notation (BPMN), 108, 112
in action, 83
diagrams, 79, 86
elements, 80
event, 84
models, 178, 183, 350, 375
notation, 79
parallel gateway element, 84
pipeline flow, 86
workflow model, 136

C

Compliance and auditability, 31, 36, 47–57, 363
Cache, 132, 225–228, 244, 245
Canary deployment, 45, 175–181, 216, 229
Carbon dioxide footprint, 74, 184, 279–281
Certificate management, 73
Certificate signing request (CSR), 17
CircleCI, 219, 221, 225, 233, 236, 315
Cloud Development Kit (CDK), 5, 361, 362, 372, 373
CloudFormation, 5, 64, 93, 361, 402
Cloud service providers (CSPs), 26, 35, 37, 68, 280, 346
Code analysis, 58, 59, 64, 97, 185, 198, 275, 314, 394
CodeDeploy, 62, 68, 179, 229
Commercial off-the-self (COTS) packages, 35
Commercial off-the-shelf (COTS), 200
clients, 200
integrity and vulnerabilities, 202
pipeline, 204
stages, 201
test/validate, application, 203
Complex event processing (CEP), 139
Compliance and auditability
application development, 48
auditability, 48
requirements analysis, 47
Compliance monitoring, 335, 347
Conditional variables, 224
Connectors/service connections, 222
Constant variables, 60
Constructs, 216
approvals, 236
auto-cancel construct, 230, 231
caching, 225–227
conditions, 224, 225
connections, 221, 222
deployment strategy, 229, 230
execution environment, 220, 221
fail fast, 232
gate, 235
matrix, 228, 229
priority, 232, 233
success/failure, 231, 232

templates and libraries, 234, 235
test shards, 233–234
triggers, 217
pipeline completed construct, 219
schedules, 218
SCM trigger, 217
webhook, 217, 218
variables, 223, 224
workflow, 236
Context diagram, 103–105, 205, 365, 366
Continuous deployment, 23, 101, 164, 194
Continuous integration/continuous delivery (CI/CD), 2, 313
ALM platforms, 16, 23
articles, 3
benefit, 12
centralized, 138
cloud, 5
concepts, 7, 8, 16
concise explanation, 13
design, 8
design patterns, 3
development, 260
diagrams, 2, 3
event-based, 189, 190, 193
foundations, 12, 146
generic, 87–89, 91, 101, 102, 108, 113, 143, 149, 212
implementing, 18
infrastructure, 2, 44
legacy, 7
migration process, 75
MQ queues, 5
naming conventions, 22
network segment, 104
philosophy, 8
and pipeline development, 75
pipelines, 2, 4, 6–8, 20, 24, 133
positioning, 13
practices, 13
promotes, 24
realization, 19
SaaS, 261
software development, 15
steps, 6
tag, 25
team effort, 19
test environment, 4
tooling, 237
validation code, 17
version, 26
vulnerabilities, 4
CPU capacity, 316, 340
Credentials/certificates, 309, 313
Cucumber directory, 372
Cucumber test, 272, 372, 387, 388
Curl command, 219
Cybersecurity, 42, 47, 316
Cycle time, 71, 343–345

D

Dangling workspaces, 45
Data anonymization, 57
Database administrator (DBA), 256

Database credentials, 45, 249, 251, 255–257
Decorator/hook, 235
Deployments, 12, 96, 165, 171
  canary, 175
  re-create, 165
  rolling update, 175
deploy.yml, 373
derive-release-version.yml, 373
Design phase, 71, 103, 375
DevOps team, 13, 30, 75, 125, 135, 186, 233, 250, 270
Docker container, 45, 59, 96, 132, 220, 221, 261
Domain-specific language (DSL) language, 210
download-artifacts.yml, 373
Download package, 201–202
Drift detection, 333, 347
Drift status, 346
Dual control, 23, 43, 100–101, 164, 275, 326
Dynamic Application Security Testing (DAST), 47, 141, 152, 153, 157
Dynamic scanning, 64, 98

E

Environment repository, 249–251
Ephemeral test environments, 35, 37, 95, 96, 153, 362
Events, 83, 352
Extended pipeline development method, 267
Extend templates, 234
External libraries, 42, 54, 132, 225, 240, 284, 365

F

Feature-based branching models, 110
Feature branch workflow model, 112
Feature flags, 257–260
Feature management, 164–165, 258–260
Federal Information Processing Standard (FIPS), 42
Full builds *vs.* incremental builds, 126, 127
Functional and nonfunctional tests, 151–153

G

Gateways, 78, 84
Gitflow, 7, 33, 117–121, 123, 124
GitHub actions, 38, 228–229
Governance, 74–75

H

Handle error task, 85
Hardware security module (HSM), 42, 251

HashiCorp Vault, 251
Healthcheck app, 360, 361, 372
Healthcheck Lambda, 361
Hotfix branches, 119

I

Implementation, 395
  application to production, 399–402
  AWSLambdaBasicExecution Role, 397
  Azure DevOps prod environment/dual control, 397, 398
  deployment strategy, 397
  libraries, 396
  mitigating action, 395
  myapp-production-deployment, 399
  quality gate, 403, 404
  requirement, 395
  test and prod environments, 398, 399
Incidents, 14, 15, 42, 194, 352–356
Include templates, 235
Infrastructure as a service (IaaS), 73, 315, 332
Infrastructure as code (IaC), 5, 95, 237, 261
Infrastructure resources, 37, 98, 141, 373
infrastructure*shaded.jar, 372
install-tools.yml, 373
Integrated development environment (IDE), 265, 288
Integration platform, 113, 130, 153, 209, 244, 250, 317
Integration server, 12, 35, 39, 45, 70, 90, 110, 264, 286
Interactive application security testing (IAST), 141, 152, 153, 157
IT value chain, 13–15

J

Jenkins, 16, 59, 104, 209–212, 251, 275
Jenkins Blue Ocean dashboard, 354
Jenkinsfile, 212, 216, 275
Jenkins freestyle project, 209, 210
Jobs, 220, 221, 223, 229, 235
JUnit tests, 289, 290, 294

K

Key performance indicators (KPIs), 68–73, 335, 336, 343, 344

L

Lead time, 71, 343–345
Life-cycle management, 16, 43
Logical design *vs.* realization
  BPMN models, 87
  CI/CD setups, 87

Long execution time *vs*. short execution time, 146
Long-lasting tests, 158
Long-running automated test, 157

M

Manageability, 59
  binary repository, 60
  deployment scripts, 60
  infrastructure code, 61
  libraries, 60
  pipeline development, 59
  run anywhere, 59
  versioning schema, 61
Manual tests, 145, 150, 153
Matrix, 43, 228, 229
Matrix Build strategy, 133
Mend Supply Chain Defender, 244
Metrics, 68
  KPIs, 68
  PKIs, 68
Microservice, 35, 36, 185–192, 237–239
Monitoring, 72
  pipelines, 73
  tools, 72
Monitoring pipelines
  business monitoring, 343–345 (*see also* Key performance indicators (KPIs)
  information sharing, 349
    events, alerts, incidents and notifications, 352–355
    notify actors stage, 350, 351
    team's branching strategy, 350
  integration platform, 335
  platform monitoring, 335, 342, 343
  security monitoring, 346–348
  systems monitoring, 335–342
Monitors, 73, 335
Multiteam build strategy, 135, 139
  environment, 134
  multiple DevOps teams, 135
Multithreaded builds, 128, 129, 304
Myapp, 360, 368, 369, 375, 376, 384, 385, 391, 394, 401, 403, 404
MyCorp.com, 359, 360, 370
mylambda.feature file, 372, 387
myLambda healthcheck, 402
myServerPool, 221

N

National Institute for the Software Industry (NISI), 18, 19, 42
Network-attached storage (NAS), 45
NexusIQ, 222, 223, 305
Nexus repository, 104, 396
Notifications, 40, 272, 312, 352–356

O

Offloaded build, 59, 130, 131
Operational pipelines, 332

AWS stacks, 334
drift detection, 333
expiration date, certificates, 333
manual operational tasks, 332
parameter, configuration service, 334
renewed certificate, 333
repeating operational function, 333
tokens/database credentials, 334
Operations tasks, 62
business continuity, 63
integration infrastructure, 63
pipelines, 63
scripts, 63
Orchestration, 60, 62, 79, 188, 320
Orchestrator, 19, 188
Organization policies, 93, 263, 276, 287, 312

P

Packaging, 23, 94
Parallel build, 128–131
Parallel execution *vs*. sequential execution, 145
Parallelize tests, 58
Performance tests, 33, 96, 97, 152, 153, 300–302, 304
Perform manual test, 30, 34, 149, 150, 155
Perform reset, 85, 86
Perform tests, 97, 185, 196, 198
Personally identifiable information (PII), 57
Pipeline code, 4, 51, 60, 64, 67, 93, 192, 221, 235, 249, 267
Pipeline Compliance Dashboard, 347, 348
Pipeline design, 186
BPMN, 80
BPMN 2.0, 79
business process modeling, 79
construct, 78
dangling, 148
manual test, 148
patterns, 79
realization cycle, 103
Pipeline development, 59
environment repository, 249
feature flags, 257–260
Git repositories, 371
MyApp, 371, 372, 374
mycorp-com, 370
pipeline creation, 375
myapp-pipeline, 375
myapp-production-deployment, 375
technical pipelines, 375
secrets management
database credentials, 255–257
retrieval, secret, 254
safest solution, 252
secret, 253
signing data, 253

Pipeline development (*cont.*)
service connections, 384
AWS and SonarCloud, 381
AWS, test and prod, 383
marketplace, 382
overview, 383
SonarCloud, 383
third-party libraries and containers, 240–244
value streams, 260
advanced pipeline development, 267, 268
application development, 264
base pipeline, 262, 268, 269
CI/CD SaaS solution, 261
compliance scanning, 263
extended pipeline development, 266
generic templates libraries, 263
pipeline code analysis, 263
pipeline generation, 270–273
pipeline of pipelines, 273–279
platform infrastructure development, 261
platform infrastructure hosting, 262
simplified pipeline development, 265, 266
specific templates/ libraries, 263
variable groups, 376
configuration, 377
generic, 377, 380
overview, 376
prod, 379, 381
sever, 378, 380
test, 378, 381
versioning and tagging, 246–248
Pipeline generator, 270–274
Pipeline implementation, 310, 311
application implementation, 319
artifact promotion, 326–328
release note, 320–325
runbook, 319, 320
integration platform
deployment tools, 315
IaaS model, 315
ISO 25010, 316
Jenkins, 315
logging, monitoring and alerting, 316
SaaS model, 315
self-hosting model, 316
SonarQube, 315
organizational impact, 311–314
pipeline code, 311
target environment preparations, 318
automated process, 318
playbook, 319
test and production environment, 318

Pipelines
caching, 132, 225–227
cross-platform, 133
dangling, 148
declarative, 211, 212
designing, 20
implementations, 116
Java-based, 8
monitoring (*see* Monitoring pipelines)
operational (*see* Operational pipelines)
pipeline of, 268, 269, 271, 273, 274, 279
regular release pipeline, 163, 164
scripted, 210–211, 224, 231
setup, 187
user interface-based, 209
use scripted/declarative, 209
YAML-based, 211, 216
Pipeline specification
complex setups, 216
constructs (*see* Constructs)
declarative pipelines, 211, 212
multibranch, multistage pipelines, 208, 209
plugins and marketplace solutions, 237
repositories, 237–240
scripted pipeline, 210
user interface-based pipelines, 209
Pipeline testing, 290, 384
analyze code stage, 385
infrastructure compliance stage, 389
integrity of artifacts, 391–394
performance and acceptance pipelines, 394
performance test, 300–305
perform test stage, 387
testability, 286–288
testing applications, 286
test specific characteristics, 286–288
unit test framework (*see* Unit testing)
pipeline-that-triggers-me, 219
PipelineUnit.java, 294
pipeline.yml files, 291, 294, 297, 373
Platform monitoring, 335, 342–343
Playbooks, 319
**/.pom pattern, 227
pom.xml file, 227, 290
Pretty Good Privacy (PGP), 244
prod-deployment.yml, 373
provision-infra.yml, 373
Provision test environment, 95–96, 101, 122, 203, 277
Public key infrastructure (PKI), 18, 68
Publish package (internal), 201, 202
Pyrsia, 244

Q

Quality assurance (QA), 64
  ALM platforms, 65
  application code, 64
  entry criteria, 66
  exit criteria, 67
  infrastructure code, 64
  pipelines, 64, 65
Quality gates, 65, 276, 287, 313, 314, 325, 326, 403–405

R

Recovery point objective (RPO), 46
Re-create deployment, 33, 101, 165–170, 173, 362
Register mitigating actions, 312
Regular deployments, 99, 194
Release build, 7, 24, 106, 349, 355
Release notes, 41, 202, 320–325
Release strategy, 3, 33, 34, 158, 159, 164, 312, 362, 367–369
Requirements analysis, 20, 29
  areas, 31
  CI/CD practices, 29
  complex pipelines, 33
  costs, 31
  feature branches short-lived, 33
  inflexibility and costs, 32
  manual testing, 34
  maturity models, 31
  microservice, 36
  principles, 29
  suggestions, 32
  way of working, 32
Requirements of myapp, 361
  manageability, 364
  security, 363
  sustainability, 361
  technology, 362
  way of working, 361
Resource, 24, 44, 48–49, 60, 101, 141, 195, 245, 334, 341, 395
Resource constraints, 31, 58–59, 184, 193–200
Road map–based release, 159–163
Rolling update deployments, 175–180
Runbook, 188, 319, 320
Runtime test-1, 311

S

Scanning complement, 64
Schedules, 218
Scripting language, 210
Secret management tools, 250
Security, 41
  ALM platform, 41
  DAST, 47
  deployment, 45
  monitoring, 346
  requirement, 46
  RTO, 46
Security pentest, 155, 156
Self-hosting, 316, 332

Semantic versioning, 62, 246, 247, 364, 369
ServiceConnectionNexusIQ service connection, 222
Shards, 233
Short feedback loops, 13, 40
Software-as-a-service (SaaS), 14, 44, 68, 153, 261, 280, 315, 331
Software delivery strategy, 102
Software development, 15–18, 20, 21, 25, 59
Software supply chain, 2, 12, 14–16, 35, 42, 68, 103, 258, 309
SonarCloud, 92, 364, 365, 381, 383, 386, 387
SonarQube, 92, 104, 196, 275, 305, 315
Source code analysis (SCA), 7, 59, 64, 184, 185, 195
Source control management system (SCM), 12, 25, 26, 43, 60, 217, 218, 246
Stage-completed.yml, 373
State, 42, 68, 106, 118, 245, 344, 352, 391
Static code scanning, 64
Sustainable computing, 74, 279
Sustainable pipeline development
  analyze code stage, 281, 282
  auto-cancel option, 281
  concept of fail fast, 281
  CSPs, 280
  rule-based trigger, 283
  SaaS ALM platform, sustainability, 280
  servers, 280
  sustainable computing, 279
  test environments, 282
  validation, 281
Systems monitoring, 335–342

T

Tagging, 51, 245–249, 272, 287, 300
Target environments, 12, 34, 53, 59–61, 104, 132, 133, 216, 311, 381
Team's branching strategy, 33, 350
Test environments, 37, 224, 259, 266, 277, 278, 282
Testing pyramid, 140, 143, 152
Test shards, 233–234
Test splitting, 233, 234
Test strategy, 3, 6, 33, 46, 139–146, 183, 285, 362
Test_task_1.2.1, 233, 234
Thinking processes, 3
Third-party libraries, 4, 43, 44, 54, 202, 240–245
Timeboxed release, 161, 162, 362, 367
Toggle feature, 258
Traceability, 24, 50, 107
Trunk-based workflow model, 105, 106
try/catch/finally construct, 231

U

Unit testing
  acceptance tests, 307
  analyze code stage, 296
  application code, 291
  Azure DevOps pipeline, 288
  compliance and security tests, 305, 306
  features, framework, 289, 290
  Java artifact, 293
  JUnit tests, 294, 297
  JUnit test2, 296
  myFeature, 296
  test approach, 299
  YAML file, 291
Unit tests, 33, 91, 112, 140, 151, 191, 264–267, 274, 288–300
update-minor.yml, 374

V

Validate entry criteria, 89–91, 108–110, 115, 121, 201, 274
Validate exit criteria stage, 99, 100, 190, 282, 325, 327, 328
Validate infrastructure compliance, 97–98, 115, 148, 278, 389, 403
Validating exit criteria, 98–100
Variables, 60, 90, 223, 224, 254, 274, 287, 373, 376
Vault, 42, 251–257, 317, 334
Versioning, 26, 61, 62, 245–249, 334, 364
Vertical scaling, 125
Vinod, 360, 363, 397

W, X

Way of working
  business organization, 32
  CI/CD process, 33
  definition, 32
  production deployment strategy, 33
  release artifacts, 33
  team's branching strategy, 33
Web application, 46
Webhook, 89, 109, 150, 217–219
WhiteSource, 244

Y

YAML-based pipelines, 211, 216

Z

zip or .tar file, 276